版权注意事项：1、书籍版权归著者和出版社所有；
2、本PDF仅用于个人获取知识，进行私底下知识交流；
3、PDF获得者不得在互联网以任何目的进行传播；
如有需要，请尽量购买正版实体书！支持书籍作者！！

Apress®

异步图书
www.epubit.com.cn

Java Threads and the Concurrency Utilities

Java线程与并发编程实践

[美] Jeff Friesen 著
鄢倩 译

- Java线程API和并发工具的实用指南

中国工信出版集团
人民邮电出版社
POSTS & TELECOM PRESS

Java Threads and
the Concurrency Utilities

Java线程与并发编程实践

[美] Jeff Friesen 著
鄢倩 译

人民邮电出版社
北京

图书在版编目（CIP）数据

Java线程与并发编程实践 / (美) 弗里森 (Jeff Friesen) 著 ; 鄢倩译. -- 北京 : 人民邮电出版社, 2017.2
ISBN 978-7-115-44036-5

Ⅰ. ①J… Ⅱ. ①弗… ②鄢… Ⅲ. ①JAVA语言－程序设计 Ⅳ. ①TP312.8

中国版本图书馆CIP数据核字(2016)第314919号

版权声明

Java Threads and the Concurrency Utilities
By Jeff Friesen, ISBN: 978-1-4842-1699-6
Original English language edition published by Apress Media.
Copyright ©2015 by Apress Media.
Simplified Chinese-language edition copyright ©2017 by Post & Telecom Press
All rights reserved.
本书中文简体字版由 Apress L.P.授权人民邮电出版社独家出版。未经出版者书面许可，不得以任何方式复制或抄袭本书内容。
版权所有，侵权必究。

◆ 著　[美] Jeff Friesen
译　鄢　倩
责任编辑　陈冀康
责任印制　焦志炜

◆ 人民邮电出版社出版发行　北京市丰台区成寿寺路 11 号
邮编　100164　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京艺辉印刷有限公司印刷

◆ 开本：800×1000　1/16
印张：17
字数：208 千字　2017 年 2 月第 1 版
印数：1 – 3 000 册　2017 年 2 月北京第 1 次印刷

著作权合同登记号　图字：01-2016-5921 号

定价：55.00 元

读者服务热线：(010)81055410　印装质量热线：(010)81055316
反盗版热线：(010)81055315

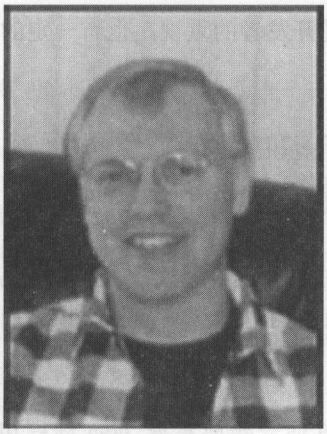
内容提要

Java 线程和并发工具是应用开发中的重要部分，备受开发者的重视，也有一定的学习难度。

本书是针对 Java 8 中的线程特性和并发工具的快速学习和实践指南。全书共 8 章，分别介绍了 Thread 类和 Runnable 接口、同步、等待和通知、线程组、定时器框架、并发工具、同步器、锁框架，以及高级并发工具等方面的主题。每章的末尾都以练习题的方式，帮助读者巩固所学的知识。附录 A 给出了所有练习题的解答，附录 B 给出了一个基于 Swing 线程的教程。

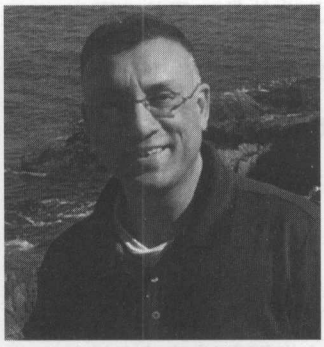
本书适合有一定基础的 Java 程序员阅读学习，尤其适合想要掌握 Java 线程和并发工具的读者阅读参考。

作者简介



Jeff Friesen 既是一名自由职业讲师，也是重点关注 Java 的软件开发者。除了是《Learn Java for Android Development》一书的作者以及《Android Recipes》一书的合著者，Jeff 还给 JavaWorld（JavaWorld.com）、informIT（InformIT.com）、Java.net 以及 DevSource（Devsource.com）写了大量关于 Java 及其他技术的文章。你可以通过他的个人网站 TutorTutor.ca 联系他。

技术审阅者简介



Sumit Pal 已经有超过 22 年的软件行业从业经验，在初创和企业级公司中担任过各类角色。他是大数据、可视化以及数据科学方面的咨询师、软件架构师，是大数据的热衷者，并且构建了端到端数据驱动分析系统。

在职业生涯的 22 年中，Sumit 陆续工作于 Microsoft（SQL Server 开发小组）、Oracle（OLAP 开发小组）以及 Verizon（大数据分析小组）公司。现在，他致力于提升各类客户的数据架构、大数据解决方案，并着手使用 Spark、Scala、Java 以及 Python 编码。他在构建从中间层、数据层到使用大数据以及 NoSQL 数据库完成可视化分析应用等可扩展系统领域有着广泛的经验。Sumit 对数据库中心、数据仓库、维度建模以及和 Java、Python，SQL 相关的数据科学方面有深刻的理解。

Sumit 拥有计算机科学的学士和硕士学位。

致谢

我要感谢在编写这本书的过程中帮助过我的所有人。尤其感谢邀请我写这本书的Steve Anglin以及写作过程中指导我的Mark Powers。

前言

线程和并发工具并非尤物，但是它们是正式应用的重要部分。本书会向你介绍 Java 8 Update 60 中线程特性以及并发工具的大部分内容。

第 1 章介绍了类 Thread 和接口 Runnable。你会学习如何创建 Thread 以及 Runnable 对象，获取和设置线程状态、启动线程、中断线程，将一条线程插入另外一条线程以及触发线程睡眠。

第 2 章关注同步。学习后你会解决一些问题，如没有同步就无法解决的竞态条件。你也能学到如何创建同步方法、块，以及如何使用忽略互斥访问的轻量级同步。

第 3 章探索等待和通知的重要话题。我们首先概览了类 Object 中支持这类概念的 API，并且通过生产者、消费者应用程序来演示这一 API，即一条线程产生条目而另一条线程消费之。

第 4 章介绍了之前 3 章没有覆盖的 3 个概念。首先，你会学习到线程组，可能想象不到它那么有用。其次，你会探索线程本地变量。最后，会学习到定时器框架，它简化了线程的定时任务。

前 4 章覆盖了低级的线程方法。第 5 章通过介绍并发工具切换至高级方法，它在简化多线程应用程序的同时也改善了性能。这一章之后还会介绍 executor、callable 以及 future。

第 6 章关注同步器（高级的同步构造）。你会学到倒计时门闩（countdown latch，一条或多条线程一直等在“门口”，直到另一条线程打开了这扇门，此时其他的线程才能继续执行）、同步屏障、交换器、信号量以及 Phaser。

第 7 章介绍了锁框架，它提供了大量针对锁及条件等待的接口和类，这种方式不同于对象原生的基于锁的同步以及 Object 的等待/通知机制。同时，它也做了一些诸如锁轮训方面的改善。

最后，第 8 章介绍了额外的并发工具，这部分内容在第 5 章到第 7 章没有涵盖。这章特别介绍了并发集合、原子变量、Fork/Join 框架以及 completion service。

每章都以各式各样的练习结尾，旨在帮助你掌握这些内容。除了问答题和判断题之外，你也经常遇到编程练习。附录 A 提供了这些问题的答案。

附录 B 提供了一个基于 Swing 线程的教程。你会学到 Swing 的单线程编程模型和大量 API，这些 API 避免了在图形上下文中使用额外线程导致的问题。你也会探索一个幻灯片应用程序，并且用这种好玩的方式结束本书。

注意： 我在某些例子中简洁地使用了 Java 8 中的 lambda 表达式，但是没有为此提供教程。你需要从其他地方获取这些知识。

注意： 你可以通过浏览器访问 www.apress.com/9781484216996，然后单击 Source Code 选项卡和紧跟其后的 Download Now 链接，来下载本书的源代码。

感谢你购买本书。我希望它对你在理解线程和并发工具上有所帮助。

——Jeff Friesen（2015 年 10 月）

目　录

第 1 部分　线程 API

第 1 章　Thread 和 Runnable · 2

1.1　Thread 和 Runnable 简介 ……2
1.1.1　创建 Thread 和 Runnable 对象 ……3
1.1.2　获取和设置线程状态 ……4
1.1.3　启动线程 ……9
1.2　操作更高级的线程任务 ……12
1.2.1　中断线程 ……12
1.2.2　等待线程 ……16
1.2.3　线程睡眠 ……20
1.3　练习 ……23
1.4　小结 ……24

第 2 章　同步 ……26

2.1　线程中的问题 ……26
2.1.1　竞态条件 ……26
2.1.2　数据竞争 ……28
2.1.3　缓存变量 ……29
2.2　同步临界区的访问 ……30
2.2.1　使用同步方法 ……32
2.2.2　使用同步块 ……33
2.3　谨防活跃性问题 ……34
2.4　volatile 和 final 变量 ……39
2.5　练习 ……48
2.6　小结 ……50

第 3 章　等待和通知 ……52

3.1　等待/通知 API 一览 ……52
3.2　生产者和消费者 ……55
3.3　练习 ……65
3.4　小结 ……66

第 4 章　额外的线程能力 ……67

4.1　线程组 ……67
4.2　线程局部变量 ……73
4.3　定时器框架 ……77
4.3.1　深入 Timer ……81
4.3.2　深入 TimerTask ……85
4.4　练习 ……87
4.5　小结 ……88

第 2 部分　并发工具类

第 5 章　并发工具类和 Executor 框架 ····· 90

5.1　并发工具类简介 ····· 90

5.2　探索 Executor ····· 91

5.3　练习 ····· 103

5.4　小结 ····· 105

第 6 章　同步器 ····· 106

6.1　倒计时门闩 ····· 106

6.2　同步屏障 ····· 111

6.3　交换器 ····· 119

6.4　信号量 ····· 126

6.5　信号量和公平策略 ····· 127

6.6　Phaser ····· 136

6.7　练习 ····· 139

6.8　小结 ····· 140

第 7 章　锁框架 ····· 142

7.1　锁 ····· 143

7.2　重入锁 ····· 145

7.3　条件 ····· 149

7.4　读写锁 ····· 157

7.5　重入读写锁 ····· 158

7.6　练习 ····· 165

7.7　小结 ····· 166

第 8 章　额外的并发工具类 ····· 167

8.1　并发集合 ····· 167

8.1.1　使用 BlockingQueue 和 ArrayBlockingQueue ····· 169

8.1.2　深入学习 Concurrent HashMap ····· 172

8.2　原子变量 ····· 173

8.3　Fork/Join 框架 ····· 179

8.4　Completion Service ····· 190

8.5　练习 ····· 194

8.6　小结 ····· 196

第 3 部分　附录

附录 A　练习题答案 ····· 198

附录 B　Swing 中的线程 ····· 225

B.1　单线程编程模型 ····· 225

B.2　线程化 API ····· 231

B.2.1　SwingUtilities 和 EventQueue ····· 231

B.2.2　SwingWorker ····· 240

B.2.3　定时器 ····· 245

B.3　基于定时器的幻灯片展示 ····· 249

第 1 部分
线程 API



第 1 章

Thread 和 Runnable

Java 程序是通过线程执行的，线程在程序中具有独立的执行路径。当多条线程执行时，它们彼此之间的路径可以不同。举个例子，一条线程可能在执行 `switch` 语句的某个 `case` 分支，另一条线程很可能在执行其他 `case` 分支。

每个 Java 应用程序都有一个执行 `main()`函数的默认主线程。应用程序也可以创建线程在后台操作时间密集型任务，以确保对用户的响应。这些封装了代码执行序列的线程对象就被称为 runnable。

Java 虚拟机给每条线程分配独立的 JVM 栈空间以免彼此干扰。独立的栈使得线程可以追踪它们自己下一条要执行的指令，这些指令会依线程不同而有所区别。栈空间也为每条线程单独准备了一份方法参数、局部变量以及返回值的拷贝。

Java 主要是通过 `java.lang.Thread` 类以及 `java.lang.Runnable` 接口实现线程机制的。本章将介绍这些类型。

1.1 Thread 和 Runnable 简介

`Thread` 类为底层操作系统的线程体系架构提供一套统一接口（操作系统通常负责创建和管理线程）。单个的操作系统线程和一个 `Thread` 对象关联。

Runnable 接口为关联 Thread 对象的线程提供执行代码。这些代码放在 Runnable 的 void run()方法中，这个方法虽然不接收任何参数且没有返回值，但是有可能抛出异常，这点我们会在第 4 章讨论。

1.1.1 创建 Thread 和 Runnable 对象

除了默认主线程，线程都是通过创建合适的 Thread 和 Runnable 对象进入应用程序的。Thread 类声明了几个构造方法来初始化 Thread 对象。其中有几个构造方法需要接收 Runnable 对象作为参数。

我们有两种方式创建 Runnable 对象。第一种方式是创建一个实现了 Runnable 接口的匿名类，如下：

```
Runnable r = new Runnable()
        {
            @Override
            public void run()
            {
                // perform some work
                System.out.println("Hello from thread");
            }
        };
```

在 Java 8 之前，这是唯一一种创建 runnable 的方式。不过，Java 8 引入 lambda 表达式更为快捷地创建 runnable：

```
Runnable r = () -> System.out.println("Hello from thread");
```

lambda 确实比匿名类更简洁。我会在本章及随后的章节继续使用这些语言特性。

注意： 一个lambda表达式是一个被传递到构造函数或者普通方法以供后续执行的匿名函数。和Runnable类似，lambda表达式以*functional interfaces*（声明单个抽象方法的接口）的形式工作。

创建Runnable对象之后，你可以把它传递到Thread类接收Runnable作为参数的构造函数中。举个例子，Thread(Runnable runnable)方法用指定的runnable初始化了一个新的Thread对象。下面的代码片段示范了这一做法：

```
Thread t = new Thread(r);
```

少数构造函数不会接收Runnable作为参数。例如，Thread()构造方法就不会接收它来初始化线程。你必须继承Thread类继而重写它的run()方法（Thread类实现了Runnable接口）并提供运行代码。如下面的代码片段所示：

```
class MyThread extends Thread
{
   @Override
   public void run()
   {
        // perform some work
        System.out.println("Hello from thread");
   }
}
// ...
MyThread mt = new MyThread();
```

1.1.2 获取和设置线程状态

一个Thread对象关联着一条线程的状态。这个状态由线程名称、线程存活的标识、线程的执行状态（是否正在执行？）、线程的优先级以及线程是否

为守护线程等标识构成。

1. 获取和设置线程的名称

每个 Thread 对象都会被赋予一个名称，这样有利于调试。这个名称如果不是显式指定的，那么默认会以一个 Thread-作为前缀。你可以通过调用 Thread 的 String getName()方法来获取这个名称。若要设置名称，则得把名称传递给一个合适的构造函数，如 Thread(Runnable r, String name)，或者调用 Thread 的 void setName(String name)方法。如下面的代码片段所示：

```
Thread t1 = new Thread(r, "thread t1");
System.out.println(t1.getName()); // Output: thread t1
Thread t2 = new Thread(r);
t2.setName("thread t2");
System.out.println(t2.getName()); // Output: thread t2
```

注意：Thread 的 long getId()方法会返回一个长整型的唯一名称。这个数字在线程的生命周期内不会改变。

2. 获取一条线程的存活状态

你可通过调用 Thread 的 boolean isAlive()方法来判断一条线程的存活状态。当线程是活的，该方法返回 true，反之，返回 false。一条线程的寿命仅仅起始于它真正在 start()方法（后面会讨论）中被启动起来，而结束于它刚刚离开 run()方法，此时线程死亡。下面的代码片段打印了一条新创建的线程的存活状态：

```
Thread t = new Thread(r);
System.out.println(t.isAlive()); // Output: false
```

3. 获取一条线程的执行状态

线程的执行状态由 Thread.State 枚举常量标识：

- NEW：该状态下线程还没有开始执行。
- RUNNABLE：该状态下线程正在 JVM 中执行。
- BLOCKED：该状态下线程被阻塞并等待一个监听锁（我会在第 2 章中介绍监听锁）。
- WAITING：该状态下线程无限期地等待另外一条线程执行特定的操作。
- TIMED_WAITING：该状态下线程在特定的时间内等待另外一条线程执行某种操作。
- TERMINATED：该状态下线程已经退出。

Thread 通过提供 Thread.State getState()方法来让应用程序判断线程的当前状态。示例如下：

```
Thread t = new Thread(r);
System.out.println(t.getState()); // Output: NEW
```

4. 获取和设置线程的优先级

当计算机有足够的处理器或处理内核，操作系统就会为每个处理器或核心分配单独的线程，这些线程可以同时执行。一旦计算机没有足够的处理器或核心的时候，多条线程只能轮转着使用共享的处理器和核心了。

注意：可以调用 java.lang.Runtime 类的 int availableProcessors() 方法，以确定 JVM 上可用的处理器或处理器核心的数量。方法的返回值可能会在 JVM 执行时发生变化，但是不会小于 1。

操作系统使用调度器（http://en.wikipedia.org/wiki/Scheduling_(computing)）来决定什么时候一个等待线程得以执行。下面的列表展示了 3 种不同的调度器：

- Linux 2.6 到 2.6.23 使用 O(1)调度器(http://en.wikipedia.org/wiki/O(1)_scheduler)。
- Linux 2.6.23 也使用了 Completely Fair 调度器(http://en.wikipedia.org/wiki/Completely_Fair_Scheduler)，它也是默认的调度器。
- Windows 基于 NT 的操作系统（如 NT、XP、Vista，以及 7）使用了多级反馈队列调度器(http://en.wikipedia.org/wiki/Multilevel_feedback_queue)，这一调度器已经被 Windows Vista 和 Windows 7 调整优化了性能。

多级反馈队列调度器和很多其他线程调度器都会考虑优先级（线程的相对重要性），通常会结合抢占式调度（高优先级线程抢占、中断并取代低优先级线程运行）和轮转时间片调度（同等优先级的线程享有同等的时间片段，也称为时间片，然后依次执行）。

注意：当探究线程究竟是并行还是并发的时候，通常会遇到两个概念。根据 Oracle 的《Multithreading Guide》(http://docs.oracle.com/cd/E19455-01/806-5257/6je9h032b/index.html)，并行是“一种发生在至少有两个线程同时执行的场景”，相反地，并发是“一种存在于至少有两个线程前进的场景，它是一种更泛化的并行，包括基于时间片的虚拟并行模式”。

Thread 通过返回当前优先级的 `int getPriority()`方法以及设置优先级的 `void setPriority(int priority)`方法来实现优先级操作。传递给优先级的值介于 `Thread.MIN_PRIORITY` 和 `Thread.MAX_PRIORITY` 之间，

而 Thread.NORMAL_PRIORITY 则确定了默认的优先级。请看下面的代码片段：

```
Thread t = new Thread(r);
System.out.println(t.getPriority());
t.setPriority(Thread.MIN_PRIORITY);
```

警告：使用 setPriority()会影响应用程序跨操作系统的可移植性，因为不同的调度器会采取不同的方式处理优先级。举个例子，一个操作系统的调度器可能会推迟低优先级线程执行，直到高优先级线程完成执行，这会导致无限延迟。而当无限期地等待执行时，低优先级线程会“饿死”，这会严重损害应用程序的性能。而其他操作系统的调度器可能不会无限期地延迟低优先级的线程，从而改善应用程序的性能。

5. 获取和设置线程的守护线程状态

Java 将线程分为守护和非守护线程。一条守护线程扮演非守护线程辅助者的角色，并且会在应用程序最后一条非守护线程消失之后自动死亡，因此应用程序才能终止。

你可以通过调用 Thread 的 boolean isDaemon()方法来判断线程是守护线程还是非守护线程，这个方法会针对一个守护线程返回 true。

```
Thread t = new Thread(r);
System.out.println(t.isDaemon()); // Output: false
```

默认情况下，和 Thread 对象关联的线程都是非守护线程。想要创建一个守护线程，你必须调用 Thread 的 void setDaemon(boolean isDaemon)方法并传入 true 作为参数。示例如下：

```
Thread t = new Thread(r);
t.setDaemon(true);
```

注意：当非守护默认主线程终止后，应用程序还会等到所有后台的非守护线程终止之后才会终止。如果后台的线程本来就是守护线程，那么当默认的主线程终止时，应用程序会立刻终止。

1.1.3 启动线程

在创建一个 Thread 对象或者其子类的对象之后，你可以通过调用 Thread 的 void start()方法启动与该对象关联的线程。如果线程之前已经启动且处于运行状态，又或者线程已经死亡，这个方法就会抛出 java.lang.IllegalThreadStateException：

```
Thread t = new Thread(r);
t.start();
```

调用 start()方法会在运行时创建底层线程，同时调度 run()方法中的指令（start()方法并不会等到所有这些任务都完成才返回）。当 run()方法执行完毕，线程就会被销毁，调用 start()方法的 Thread 对象不再可用，这也是 start()方法会导致 IllegalThreadStateException 的原因。

我已经创建了一个应用，它包含了从线程创建到启动的多种基本示例。请查看清单 1-1。

清单 1-1 线程基础示例

```
public class ThreadDemo
{
    public static void main(String[] args)
    {
        boolean isDaemon = args.length != 0;
```

```
Runnable r = new Runnable()
             {
               @Override
               public void run()
               {
                  Thread thd = Thread.currentThread();
                  while (true)
                  System.out.printf("%s is %salive and in %s " +
                                    "state%n",
                                    thd.getName(),
                                    thd.isAlive() ? "" : "not ",
                                    thd.getState());
               }
             };
Thread t1 = new Thread(r, "thd1");
if (isDaemon)
   t1.setDaemon(true);
System.out.printf("%s is %salive and in %s state%n",
                  t1.getName(),
                  t1.isAlive() ? "" : "not ",
                  t1.getState());
Thread t2 = new Thread(r);
t2.setName("thd2");
if (isDaemon)
    t2.setDaemon(true);
System.out.printf("%s is %salive and in %s state%n",
                  t2.getName(),
                  t2.isAlive() ? "" : "not ",
                  t2.getState());
```

```
            t1.start();
            t2.start();
        }
    }
```

首先，默认的主线程基于命令行参数是否存在来初始化 isDaemon 变量。只要有一个参数传递过来，isDaemon 就会设置成 true。否则，设置为 false。

其次，一个 runnable 对象被创建了。这个对象首先调用 Thread 的静态方法 Thread currentThread()获得当前执行线程关联的 Thread 对象的引用。该引用继而会用于获取该线程的相关信息，这些信息最终会被打印出来。

到这里，初始化为上面的 runnable 且名为 thd1 的 Thread 对象就被创建好了。如果 isDaemon 是 true，那么这个线程对象就会标记成守护线程，它的名字、存活状态以及执行状态随后会被打印。

第二个初始化为 runnable 且名为 thd2 的线程对象也被创建好了。与之前相同，如果 isDaemon 是 true，该线程对象会标记成守护线程，它的名字、存活状态以及执行状态也会被打印出来。

最后，两条线程都启动起来。下面来编译清单 1-1：

```
javac ThreadDemo.java
```

运行程序：

```
java ThreadDemo
```

以下是我在 64 位的 Windows 7 操作系统上观测到的不断打印的结果中最开始的部分：

```
thd1 is not alive and in NEW state
thd2 is not alive and in NEW state
thd1 is alive and in RUNNABLE state
```

```
thd2 is alive and in RUNNABLE state
```

你很可能在你的操作系统看到不同的输出结果。

提示：为了终止无限运行的应用程序，可以在 Windows 或者非 Windows 系统上同时按住 Ctrl 和 C 键。

现在，运行程序：

```
java ThreadDemo x
```

和前面所有非守护线程执行的结果不同，命令行参数的存在导致所有的线程都会作为守护线程执行。因此，这些线程执行到默认主线程终止。你应该能观测到更简略的打印结果。

1.2 操作更高级的线程任务

之前的线程任务都和如何配置一个线程对象以及启动关联的线程相关。不过，Thread 类也能支持更多高级的任务，包括中断其他线程、将线程 join 到另一条线程中以及致使线程睡眠。

1.2.1 中断线程

`Thread` 类提供了一种线程可以中断其他线程的机制。当一个线程被中断时，它会抛出 `java.lang.InterruptedException`。这一机制由下面的 3 种方法构成。

- `void interrupt()`：中断调用此方法的 Thread 对象所关联的线程。

当一条线程由于调用了 Thread 的 sleep()或者 join()方法（这一章后面会讨论到）而被阻塞住时，该线程的中断状态就会被清除，同时抛出 InterruptedException。否则，除了会设置中断状态，其他动作也会取决于当前线程的行为相应地发生（详情参见 JDK 文档）。

- static boolean interrupted()：验证当前线程是否已经中断，在这个例子中会返回 true。该线程的中断状态会被这个方法清除掉。
- boolean isInterrupted()：验证线程是否已经中断，这个例子中会返回 true。该线程的中断状态不受此方法的影响。

我创建了一个应用程序来演示线程中断，见清单 1-2。

清单 1-2 线程中断示例

```
public class ThreadDemo
{
  public static void main(String[] args)
  {
    Runnable r = new Runnable()
                 {
                   @Override
                   public void run()
                   {
                     String name = Thread.currentThread().getName();
                     int count = 0;
                     while (!Thread.interrupted())
                       System.out.println(name + ": " + count++);
                   }
                 };
    Thread thdA = new Thread(r);
```

```
        Thread thdB = new Thread(r);
        thdA.start();
        thdB.start();
        while (true)
        {
            double n = Math.random();
            if (n >= 0.49999999 && n <= 0.50000001)
                break;
        }
        thdA.interrupt();
        thdB.interrupt();
    }
}
```

默认的主线程首先创建runnable对象，用于获取当前线程的名称。这个runnable对象随后声明了一个计数变量并且进入到`while`循环中，重复打印线程名称和计数变量的值，同时不断递增计数变量的值，直到该线程被中断。

接下来，默认的主线程创建了一对`Thread`对象，它们执行runnable并启动这些后台线程。

为了给这些后台线程一些时间以便在中断之前打印几条消息，默认主线程进入了一个基于`while`的忙循环，该循环语句就是拿来消耗一些时间的。这个循环会重复地获取随机数直到数字落入一段狭窄的区间内。

注意：因为会浪费处理器时间，忙循环不是一个好主意。在本章后面我会展示一个更好的解决方案。

while 循环终止之后，默认的主线程在每个后台的线程对象上执行 interrupt()方法。每个后台线程在下一次执行 Thread.interrupted()时，会返回 true 并且同时终止循环。

编译清单 1-2（javac ThreadDemo.java），并运行最终程序（java ThreadDemo）。你应该能看到包含递增计数变量的消息在 Thread-0 和 Thread-1 之间交替。示例如下：

```
Thread-1: 67
Thread-1: 68
Thread-0: 768
Thread-1: 69
Thread-0: 769
Thread-0: 770
Thread-1: 70
Thread-0: 771
Thread-0: 772
Thread-1: 71
Thread-0: 773
Thread-1: 72
Thread-0: 774
Thread-1: 73
Thread-0: 775
Thread-0: 776
Thread-0: 777
Thread-0: 778
Thread-1: 74
Thread-0: 779
Thread-1: 75
```

1.2.2 等待线程

线程（如默认的主线程）会偶尔启动另一个线程去操作单调的计算、下载大文件或者操作一些其他的耗时任务。在结束它自己的任务之后，这个启动工作线程的线程就准备着处理工作线程的结果，同时等待该工作线程“寿终正寝”。

Thread 类提供了 3 种 join()方法，允许调用线程等待执行此方法的线程对象所关联的线程执行完毕。

- void join()：无限期地等待直至该线程死亡。当任意线程中断当前线程的时候，InterruptedException 就会抛出。如果该异常被抛出，该线程的中断状态就会被清除。
- void join(long millis)：该线程死亡之前最多等待 millis 毫秒。如果传递 0 作为参数就会无限期地等待——join()其实就调用了 join(0)方法。如果 millis 是负数，那么就会导致 IllegalArgumentException 被抛出。当任意线程中断了当前线程，就会导致 InterruptedException 被抛出，如果该异常被抛出，该线程的中断状态会被清除。
- void join(long millis, int nanos)：该线程死亡之前最多等待 millis 毫秒加 nanos 纳秒。当 millis 是负数、nanos 是负数或者 nanos 大于 999999 的时候，会导致 IllegalArgumentException 被抛出。当任意线程中断了当前线程，就会导致 InterruptedException 被抛出，如果该异常被抛出，该线程的中断状态会被清除。

为了演示不含参数的 join()方法，我创建了一个应用程序来计算数学中的常量 pi 到小数点后 50000 位。它是根据 17 世纪早期的一位英国数学家 John Machin（https://en.wikipedia.org/wiki/John_Machin）发明的算法来计

算的。这一算法首先计算 pi/4 = 4 × arctan(1/5)−arctan(1/239)，然后把结果乘以 4 得到 pi 的值。因为反正切函数使用了幂级的条件计算，条件越多，pi 的值会越精确（从到小数点后多少位这方面来看）。清单 1-3 展示了源代码。

清单 1-3　演示 Thread Joining

```
import java.math.BigDecimal;

public class ThreadDemo
{
   // constant used in pi computation

   private static final BigDecimal FOUR = BigDecimal.valueOf(4);

   // rounding mode to use during pi computation

   private static final int roundingMode = BigDecimal.ROUND_HALF_EVEN;

   private static BigDecimal result;

   public static void main(String[] args)
   {
      Runnable r = () ->
                   {
                      result = computePi(50000);
                   };
      Thread t = new Thread(r);
      t.start();
      try
```

```
        {
            t.join();
        }
        catch (InterruptedException ie)
        {
            // Should never arrive here because interrupt() is never
            // called.
        }
        System.out.println(result);
    }
    /*
     * Compute the value of pi to the specified number of digits after the
     * decimal point. The value is computed using Machin's formula:
     *
     * pi/4 = 4*arctan(1/5)-arctan(1/239)
     *
     * and a power series expansion of arctan(x) to sufficient precision.
     */
    public static BigDecimal computePi(int digits)
    {
      int scale = digits + 5;
      BigDecimal arctan1_5 = arctan(5, scale);
      BigDecimal arctan1_239 = arctan(239, scale);
      BigDecimal pi = arctan1_5.multiply(FOUR).
                      subtract(arctan1_239).multiply(FOUR);
      return pi.setScale(digits, BigDecimal.ROUND_HALF_UP);
     }
    /*
```

```
 * Compute the value, in radians, of the arctangent of the inverse of
 * the supplied integer to the specified number of digits after the
 * decimal point. The value is computed using the power series
 * expansion for the arc tangent:
 *
 * arctan(x) = x-(x^3)/3+(x^5)/5-(x^7)/7+(x^9)/9 ...
 */

public static BigDecimal arctan(int inverseX, int scale)
{
   BigDecimal result, numer, term;
   BigDecimal invX = BigDecimal.valueOf(inverseX);
   BigDecimal invX2 = BigDecimal.valueOf(inverseX * inverseX);
   numer = BigDecimal.ONE.divide(invX, scale, roundingMode);
   result = numer;
   int i = 1;
   do
   {
       numer = numer.divide(invX2, scale, roundingMode);
       int denom = 2 * i + 1;
       term = numer.divide(BigDecimal.valueOf(denom), scale,
                           roundingMode);
       if ((i % 2) != 0)
            result = result.subtract(term);
       else
            result = result.add(term);
       i++;
   }
```

```
            while (term.compareTo(BigDecimal.ZERO) != 0);
            return result;
        }
    }
```

默认的主线程首先创建了一个 runnable 去计算 pi 到小数点后 50000 位，然后把结果赋值给名为 `result` 的 `java.math.BigDecimal` 对象。为了代码简洁，这里使用了 lambda 表达式。

这个线程随后创建了一个 `Thread` 对象去执行 runnable 并启动了一个工作线程来执行操作。

这里，默认的主线程在该 `Thread` 对象上调用了 `join()`方法等待工作线程死亡。当工作线程死亡了，默认主线程会打印出 `BigDecimal` 对象的值。

编译清单 1-3 的代码（`javac ThreadDemo.java`）并运行最终程序（`java ThreadDemo`）。我观察到的前段部分输出如下：

```
3.14159265358979323846264338327950288419716939937510582097494459230781640628
62089986280348253421170679821480865132823066470938446095505822317253594081
28481117450284102701938521105559644622948954930381964428810975665933446128 4
75648233786783165271201909145648566923460348610454326648213393607260249141 2
73724587006606315588174881520920962829254091715364367892590360011330530548 8
20466521384146951941511609433057270365759591953092186117381932611793105118 5
48074462379962749567351885752724891227938183011949129833673362440656643086
02139494639522473719070217986094370277053921717629317675238467481846766940
5132000568127
```

1.2.3 线程睡眠

`Thread` 类声明了一对静态方法致使线程睡眠（暂时性地停止执行）。

- void sleep(long millis)：睡眠 millis 毫秒数。线程睡眠的实际的毫秒数取决于系统定时器和调度器的精度。如果 millis 是负数，那么就会导致 IllegalArgumentException 被抛出。当任意线程中断了当前线程，就会导致 InterruptedException 被抛出，如果该异常被抛出，该线程的中断状态会被清除。
- void sleep(long millis, int nanos)：睡眠 millis 毫秒数和 nanos 纳秒数。实际睡眠的毫秒数和纳秒数取决于系统定时器和调度器的精度。当 millis 是负数，nanos 是负数或者 nanos 大于 999999 的时候，会导致 IllegalArgumentException 被抛出。当任意线程中断了当前线程，就会导致 InterruptedException 被抛出，如果该异常被抛出，该线程的中断状态会被清除。

sleep()方法相较于忙循环更好，因为它们不会浪费处理器周期。

我已经重构了清单 1-2 的应用程序来展示线程睡眠。请看清单 1-4。

清单 1-4　线程睡眠示例

```
public class ThreadDemo
{
    public static void main(String[] args)
    {
        Runnable r = new Runnable()
                     {
                         @Override
                         public void run()
                         {
                             String name = Thread.currentThread().getName();
```

```
                int count = 0;
                while (!Thread.interrupted())
                    System.out.println(name + ":" + count++);
            }
        };
        Thread thdA = new Thread(r);
        Thread thdB = new Thread(r);
        thdA.start();
        thdB.start();
        try
        {
            Thread.sleep(2000);
        }
        catch (InterruptedException ie)
        {
        }
        thdA.interrupt();
        thdB.interrupt();
    }
}
```

清单 1-2 和清单 1-4 唯一的不同之处就是使用 `Thread.sleep(2000)`替代了忙循环，睡眠了 2 秒。

编译清单 1-4（`javac ThreadDemo.java`），运行最终程序（`java ThreadDemo`）。由于睡眠时间是大概的时间，所以在多次运行中打印出的行数会有差异。但是，这种差异不会特别大。举个例子，你不会在某次运行看到 10 行，而在另外一次运行中看到 1000 万行。

1.3 练习

接下来的练习旨在测试你对第 1 章内容的掌握程度。

1. 给出线程的定义。

2. 给出 runnable 的定义。

3. Thread 类和 Runnable 接口完成了什么？

4. 指出创建一个 Runnable 对象的两种方式。

5. 指出关联一个 runnable 到一个 Thread 对象的两种方式。

6. 指出 5 种 Thread 的状态。

7. 判断对错：默认线程的名字会以 Thd-作为前缀。

8. 如何给线程赋予非默认的名称？

9. 如何确定线程是死是活？

10. 指出 Thread.State 枚举的所有常量。

11. 如何获取当前线程的执行状态？

12. 给出优先级的定义。

13. 如何通过 setPriority()来影响应用程序跨操作系统的可移植性？

14. 确定 Thread 的 void setPriority(int priority)方法参数的取值范围。

15. 判断对错：当应用程序的最后一个非守护线程死亡之后，守护线程也会自动死亡以使得应用程序退出。

16. 在一个正在运行或者已经死亡的 Thread 对象上调用 Thread 的 void start()方法会发生什么？

17. 如何在 Windows 上停止一个无法终止的程序？

18. 确定组成 Thread 中断机制的所有方法。

19. 判断对错：boolean isInterrupted()方法清除了线程的中断状态。

20. 当线程中断了，该线程会如何反映？

21. 给出忙循环的定义。

22. 指出让一条线程等待另一个线程直至死亡的 Thread 方法。

23. 指出让一条线程睡眠的 Thread 方法。

24. 编写一个名为 IntSleep 的应用程序，其创建一条后台线程，不断地打印出 Hello，之后睡眠 100ms。在睡眠了 2s 之后，默认主线程应该中断后台线程，此线程在打印出 interrupted 之后跳出循环。

1.4　小结

Java 应用程序通过线程执行，线程在程序代码中具有独立的执行路径。每个 Java 应用程序都有一个执行 main()函数的默认主线程。应用程序也可以创建线程在后台操作时间密集型任务以确保对用户的响应。这些封装了代码执行序列的线程对象就称为 runnables。

Thread 类为底层操作系统的线程体系架构提供一套统一接口（操作系统通常负责创建和管理线程）。单个的操作系统线程关联为一个 Thread 对象。

Runnable 接口为关联 Thread 对象的线程提供执行代码。这些代码放在 Runnable 的 void run()方法中，尽管它不接收任何参数且没有返回值，但有可能抛出异常。

除了默认的主线程，线程都是通过创建合适的 Thread 和 Runnable 对象而引入应用程序的。Thread 类声明了几个构造方法来初始化 Thread 对象。其中有几个构造方法需要接收 Runnable 对象作为参数。

一个 Thread 对象关联着一条线程的状态。这个状态由线程名称、线程存活的标识、线程的执行状态（是否正在执行？）、线程的优先级以及线程是否为守护线程等标识构成。

在创建一个 Thread 对象或者其子类的对象之后，你可以通过调用 Thread 的 void start()方法启动与该对象关联的线程。如果线程之前已经启动且处于运行状态，又或者线程已经死亡，这个方法就会抛出 java.lang.IllegalThreadStateException。

之前的线程任务都和如何配置一个线程对象以及启动关联的线程相关。不过，Thread 类也能支持更多高级的任务，包括中断其他线程、将线程 join 到另一条线程中以及致使线程睡眠。

第 2 章会涉及与同步相关的话题。

第2章

同步

线程交互通常是通过共享变量完成的，当线程之间没有交互时，开发多线程的应用程序会变得简单许多。一旦发生了交互，很多诱发线程不安全（在多线程环境下不正确）的因素就会暴露出来。在这一章中，你将会认识到这些问题，同时也会学习如何正确地使用 Java 面向同步的特性来克服它们。

2.1 线程中的问题

Java 对线程的支持促进了响应式、可扩展应用程序的发展。不过，这样的支持是以增加复杂性作为代价的。如果不多加小心，你的代码就会到处充斥着极难以察觉的 bug，而这些 bug 多和竞态条件、数据竞争以及缓存变量有关。

2.1.1 竞态条件

当计算的正确性取决于相对时间或者调度器所控制的多线程交叉时，竞态条件就会发生。下面的代码片段描述了只要满足一个特定的前置条件，就会触发计算的场景：

```
if (a == 10.0)
   b = a / 2.0;
```

在单线程的环境中，这段程序没有任何问题。在多线程环境下，如果 a 和 b 都是局部变量，那么也没有问题。但是，假设 a 和 b 是实例变量或者类(static)变量，并且有两条线程同时访问这段代码，就有问题了。

假设一条线程已经执行完 `if (a == 10.0)`，在即将执行 `b = a / 2.0` 时，被调度器暂停了，与此同时，调度器恢复了另一条线程改变了 a 的值；当前一条线程恢复执行，变量 b 却不会等于 5.0（如果 a 和 b 是局部变量，因为每个线程都会有自己的局部变量拷贝，所以竞态条件不会发生）。

这段代码就是竞态条件中称为 check-then-act 的一个经典例子。在这种竞态条件下，很可能会用过时的观测状态来决定下一步的动作。在前面的代码片段中，“检查”是 `if (a == 10.0)`，“动作”则是 `b = a / 2.0;`。

另外一种类型的竞态条件就是 read-modify-write，这种情况下，新状态继承自旧状态。旧状态被读取，然后更改，最后更新，通过这 3 个不可分割的操作来得到更改后的结果。只不过，这些操作的组合并非不可分割。

典型的 read-modify-write 的例子就是用一个递增的变量来生成唯一的数字标识。在下面的代码片段中，假设 counter 变量是一个类型为 int（初始化为 1）的实例变量，两条线程同时访问这段代码：

```
public int getID()
{
   return counter++;
}
```

尽管看上去这是个单一操作，但事实上，表达式 counter++是 3 个单独的操作：读取 counter 的值，给值加 1，然后把更新之后的值存储到 counter 中。当时读取的值就是整个表达式的返回值。

假设在被调度器阻断之前，线程 1 调用了 getID()方法，同时读取了 counter 的值，此时其值是 1。现在，假设线程 2 运行，调用了 getID()方法，读取了 counter 的值（1），对这个值加 1，把结果（2）存储到 counter 中，然后将 1 返回给调用者。

在这种情况下，假设线程 1 恢复过来了，对之前读到的值（1）加 1，然后把结果（2）存储到 counter 变量中，然后将 1 返回给调用者。由于线程 1 撤销了线程 2 的动作，我们就会错过一次递增并生成了一个重复的 ID。所以这个方法是无效的。

2.1.2 数据竞争

竞态条件经常会和数据竞争相混淆。数据竞争指的是两条或两条以上的线程（在单个应用中）并发地访问同一块内存区域，同时其中至少有一条是为了写，而且这些线程没有协调对那块内存区域的访问。当满足这些条件的时候，访问顺序就是不确定的。依据这种顺序，每次运行都可能会产生不同的结果。看下面的例子：

```
private static Parser parser;

public static Parser getInstance()
{
   if (parser == null)
      parser = new Parser();
   return parser;
}
```

假设线程 1 首先调用了 getInstance()方法。由于它检测到属性 parser 是空值，线程 1 就会实例化 Parser 并且将引用赋给变量 parser。随后，当线

程 2 调用 getInstance()方法时，它可能检测到 parser 已经包含了一个非空的引用，于是简单地返回了 parser 的值；另一种可能是，线程 2 检测到 parser 的值仍然是空，于是创建了一个新的 Parser 的对象。由于线程 1 写 parser 变量和线程 2 读 parser 变量之间没有 happens-before ordering（一个动作先于另一个动作发生）的保证（这里不存在对 parser 访问顺序的协同），数据竞争产生了。

2.1.3 缓存变量

为了提升性能，编译器 Java 虚拟机（JVM）以及操作系统会协调在寄存器中或者处理器缓存中缓存变量，而不是依赖主存。每条线程都会有其自己的变量拷贝。当线程写入这个变量的时候，其实是写入自己的拷贝；其他线程不太可能在看到自己的变量拷贝发生更改。

第 1 章给出的 ThreadDemo 的应用程序（参见清单 1-3）暴露了这个问题。这里我重新列出部分源码以供参考：

```
private static BigDecimal result;

public static void main(String[] args)
{
   Runnable r = () ->
                {
                   result = computePi(50000);
                };
   Thread t = new Thread(r);
   t.start();
   try
```

```
        {
            t.join();
        }
        catch (InterruptedException ie)
        {
            // Should never arrive here because interrupt() is never
            // called.
        }
        System.out.println(result);
    }
```

类属性 result 示范了缓存变量的问题。该属性在 lambda 表达式的上下文当中被一条工作线程访问并执行代码 result = computePi(50000);，然后默认主线程执行 System.out.println(result);。

这个工作线程能够将 computePi()的返回值存储到自己的 result 变量的拷贝中。默认主线程很可能无法看到 result = computePi(50000);的赋值，并且它的本地拷贝会保持原来默认的 null 值。这个 null 值会取代 result 的字符串表示（即计算好的 pi 的值）被打印出来。

2.2 同步临界区的访问

你会看到，同步能够解决之前的线程问题。同步是 JVM 的一个特性，旨在保证两个或者多个并发的线程不会同时执行同一块临界区，临界区就是必须以串行方式（一次一条线程）访问的一段代码块。

因为其他线程在临界区中的时候每条线程对该临界区的访问都会互斥地执行，这种同步属性就称为互斥。由于这个原因，线程获取到的锁经常称为互斥锁。

同步也表现出可见性，该属性能够保证一条线程在临界区执行的时候总是能看到共享变量最近的修改。当进入临界区时，它从主存中读入这些变量，离开时把这些变量的值写入主存。

同步是通过监听器来实现的，监听器是针对临界区构建的并发访问控制，并发必须以不可分割的形式执行。每一个 Java 对象都和一个监听器相关联，这样线程就可以通过获取和释放监听器的锁（一个标识）来上锁和解锁。

注意： 当调用 Thread 任意的 sleep()方法时，已经获取锁的线程不会释放锁。

只有一个线程可以持有监听器的锁，任意尝试锁住该监听器的线程都会一直阻塞，直到能够获取锁为止。当线程离开临界区，它会通过释放锁来解锁监听器。

为了防止发生死锁（后面会讨论），锁设计成可重入的。当线程尝试获取它已经持有的锁，请求会成功。

提示： java.lang.Thread 类声明了一个 static boolean holdsLock (Object o)方法，即当调用线程持有对象 o 的锁，该方法会返回 true。你会很容易在断言语句中找到这个方法，例如：assert Thread.holdsLock(o);。

Java 提供 synchronized 关键字来串行线程对方法和语句块（临界区）的访问。

2.2.1 使用同步方法

同步方法会在方法头部包含 synchronized 关键字。举个例子，你可以使用这一关键字来同步之前的 getID()方法，像下面这样来克服 read-modify-write 竞态条件：

```
public synchronized int getID()
{
   return counter++;
}
```

当同步在实例方法上，锁会和调用该方法的实例对象关联。举个例子，参考下面的 ID 类：

```
public class ID
{
   private int counter; // initialized to 0 by default

   public synchronized int getID()
   {
      return counter++;
   }
}
```

假设你指定了下面的代码序列：

```
ID id = new ID();
System.out.println(id.getID());
```

锁和 ID 对象相关联，对象的引用存储在 id 变量中。如果其他线程在该方法执行过程中调用 id.getID()方法，这些线程不得不等待正在执行的线程释放锁。

当同步在一个类方法上时，锁会和调用该类方法的类所对应的 java.lang.Class 对象相关联。举个例子，参考下面的 ID 类：

```
public class ID
{
   private static int counter; // initialized to 0 by default

   public static synchronized int getID()
   {
      return counter++;
   }
}
```

假设你指定了下面的代码序列：

```
System.out.println(ID.getID());
```

锁和 ID 类关联的 Class 对象 ID.class 相关联。如果其他线程在该方法执行过程中调用 ID.getID()方法，这些线程也不得不等待正在执行的线程释放掉锁。

2.2.2 使用同步块

一个同步块语句把这个待锁住的对象作为前缀头。它具有以下语法结构：

```
synchronized(object)
{
   /* statements */
}
```

从这个语法来看，object 是某个对象的引用。锁和该对象相关联。

我之前列出第 1 章中一段遭遇缓存变量的程序。你可以通过两个同步块解决这一问题。

```
Runnable r = () ->
             {
                synchronized(FOUR)
                {
                   result = computePi(50000);
                }
             };
   // …
synchronized(FOUR)
{
   System.out.println(result);
}
```

这两个同步块标识了一对临界区。每个同步块被同一个对象所保护，以致于同一时间只能有一条线程能在这些同步块中执行。每条线程在进入它的临界区之前，必须获取被常量 FOUR 所引用对象的锁。

这段代码块打开了一个关于同步块和同步方法很重要的点。要么访问同一段代码序列的两条或两条以上线程必须获取同一个锁，要么不存在同步。这也暗示着（进入临界区）必须访问同一个对象。在之前的例子中，活跃性 FOUR 被放到两个位置，以致于同时只能有一条线程出现在其中一个临界区。如果我在一个位置指定 synchronized(FOUR)，在另一个地方指定 synchronized("ABC")，由于涉及两个不同的锁，也就不存在同步了。

2.3 谨防活跃性问题

活跃性这个词代表着某件正确的事情最终会发生。活跃性失败发生在应用

程序触及一种无法继续执行的状态。在单线程的应用程序中，无限循环就是一个例子。多线程应用程序面临着诸如死锁、活锁和饿死的额外挑战。

- 死锁：线程 1 等待线程 2 互斥持有的资源，而线程 2 也在等待线程 1 互斥持有的资源。两条线程都无法继续执行。
- 活锁：线程 x 持续重试一个总是失败的操作，以致于无法继续执行。
- 饿死：线程 x 一直被（调度器）延迟访问其赖以执行的资源。或许是调度器先于低优先级的线程执行高优先级的线程，而总是有一个高优先级的线程可以执行。饿死通常也称为无限延迟。

死锁会发生在 synchronized 关键字带来的过多同步上。如果不小心，你可能就会遭遇锁同时被多条线程竞争的情形；即线程自身缺失继续执行的锁，却持有其他线程需要的锁，同时由于其他线程持有临界区的锁，导致没有一条线程能够通过临界区，进而释放自己所持有的锁。清单 2-1 就是描述该场景的一个典型例子。

清单 2-1　一个死锁的问题

```
public class DeadlockDemo
{
   private final Object lock1 = new Object();
   private final Object lock2 = new Object();

   public void instanceMethod1()
   {
      synchronized(lock1)
      {
         synchronized(lock2)
         {
```

```
            System.out.println("first thread in instanceMethod1");
            // critical section guarded first by
            // lock1 and then by lock2
        }
    }
}
public void instanceMethod2()
{
    synchronized(lock2)
    {
        synchronized(lock1)
        {
            System.out.println("second thread in instanceMethod2");
            // critical section guarded first by
            // lock2 and then by lock1
        }
    }
}

public static void main(String[] args)
{
    final DeadlockDemo dld = new DeadlockDemo();
    Runnable r1 = new Runnable()
                {
                    @Override
                    public void run()
                    {
                        while(true)
                        {
                            dld.instanceMethod1();
```

```
                try
                {
                    Thread.sleep(50);
                }
                catch (InterruptedException ie)
                {
                }
            }
        }
    };
Thread thdA = new Thread(r1);
Runnable r2 = new Runnable()
        {
            @Override
            public void run()
            {
                while(true)
                {
                    dld.instanceMethod2();
                    try
                    {
                        Thread.sleep(50);
                    }
                        catch (InterruptedException ie)
                        {
                        }
                }
            }
        };
```

```
        Thread thdB = new Thread(r2);
        thdA.start();
        thdB.start();
    }
}
```

清单2-1中线程A和线程B在不同的时间分别调用了instanceMethod1()和instanceMethod2()方法。参考下面的执行序列：

（1）线程A调用instanceMethod1()，获取到lock1引用对象的锁，然后进入它外部的临界区（但是还没有获取lock2引用对象的锁）。

（2）线程B调用instanceMethod2()，获取到lock2引用对象的锁，然后进入它外部的临界区（但是还没有获取lock1引用对象的锁）。

（3）线程A尝试去获取和lock2相关联的锁。JVM强制线程在内部临界区之外等待，由于线程B持有那个锁。

（4）线程B尝试去获取和lock1相关联的锁。JVM强制线程在内部临界区之外等待，由于线程A持有那个锁。

（5）由于其他线程持有了必要的锁，没有一条线程能继续执行。遭遇死锁，程序（至少在这两条线程的上下文中）就冻结住了。

照下面那样编译清单2-1：

```
javac DeadlockDemo.java
```

运行程序：

```
java DeadlockDemo
```

你应该能在标准输出流中观测到交替打印的first thread in instanceMethod1和second thread in instanceMethod2信息，直到程序因死锁而冻结。

尽管前面的例子很清晰地识别了死锁的状态，但侦测死锁还是不容易。举个例子，你的代码可能在多个类中（在多个源文件里）包含如下的环形关系：

- 类 A 的同步方法调用了类 B 的同步方法。
- 类 B 的同步方法调用了类 C 的同步方法。
- 类 C 的同步方法调用了类 A 的同步方法。

如果线程 A 调用了类 A 的同步方法，线程 B 调用了类 C 的同步方法，因为线程 A 还在那个方法当中，当线程 B 尝试调用类 A 的同步方法时，会被阻塞住。线程 A 会继续执行直至其调用类 C 的同步方法，然后阻塞住。死锁发生了。

注意：Java 语言和 JVM 都没有提供一种方式来避免死锁，所以这一任务就落到你的身上。避免死锁最简单的方式，就是阻止同步方法或者同步块调用其他的同步方法和同步块。尽管这个建议能避免死锁发生，但还是不现实，因为，在你的某一个同步方法和同步块中，很可能需要调用 Java API 中的同步方法，而且这个建议有点因噎废食，因为被调用的同步方法和同步块很可能不会调用其他的同步方法和同步块，从而不会导致死锁发生。

2.4 volatile 和 final 变量

你之前学到的同步展示了两种属性：互斥性和可见性。`synchronized` 关键字与两者都有关系。Java 同时也提供了一种更弱的、仅仅包含可见性的同步形式，并且只以 `volatile` 关键字关联。

假设你自己设计了一个停止线程的机制（因为无法使用 Thread 不安全的 stop()方法））。清单 2-2 中 ThreadStopping 程序源码展示了该如何完成这项任务。

清单 2-2 尝试停止一个线程

```
public class ThreadStopping
{
   public static void main(String[] args)
   {
      class StoppableThread extends Thread
      {
         private boolean stopped; // defaults to false

         @Override
         public void run()
         {
            while(!stopped)
               System.out.println("running");
         }

         void stopThread()
         {
            stopped = true;
         }
      }
      StoppableThread thd = new StoppableThread();
      thd.start();
      try
      {
```

```
            Thread.sleep(1000); // sleep for 1 second
        }
        catch (InterruptedException ie)
        {
        }
        thd.stopThread();
    }
}
```

清单 2-2 中的 main()方法声明了一个叫做 StoppableThread 的本地类，它继承自 Thread。在初始化完 StoppableThread 之后，默认的主线程启动和这个 Thread 对象关联的线程。之后它睡眠 1 秒，并且在死亡之前调用 StoppableThread 的 stop()方法。

StoppableThread 声明了一个被初始化为 false 的 stopped 实例变量，stopThread()方法会将该变量设置为 true，同时 run()方法中的 while 循环会在每次迭代中检查 stopped 的值是否已经修改为 true。

照下面编译清单 2-2：

```
javac ThreadStopping.java
```

运行程序：

```
java ThreadStopping
```

你应该能观测到一系列运行时的消息。

当你在单处理器/单核的机器上运行这个程序的时候，很可能会观测到程序停止。但是在一个多处理器的机器或多核单处理器的机器上，可能就看不到程序停止，因为每个处理器或者核心很可能有自己的一份 stopped 的拷贝，当一条线程修改了自己的拷贝，其他线程的拷贝并没有被改变。

你或许决定使用synchronized关键字以确保只能访问主存中的stopped变量。然后经过一番思考，你决定在清单2-3中使用同步访问一对临界区的方式来解决这个问题。

清单2-3 尝试使用synchronized来停止一个线程

```
public class ThreadStopping
{
   public static void main(String[] args)
   {
      class StoppableThread extends Thread
      {
         private boolean stopped; // defaults to false

         @Override
         public void run()
         {
            synchronized(this)
            {
               while(!stopped)
                  System.out.println("running");
            }
         }

         synchronized void stopThread()
         {
            stopped = true;
         }
      }
```

```
        StoppableThread thd = new StoppableThread();
        thd.start();
        try
        {
            Thread.sleep(1000); // sleep for 1 second
        }
        catch (InterruptedException ie)
        {
        }
        thd.stopThread();
    }
}
```

出于两个因素考虑，清单 2-3 不是一个好主意。尽管你只需解决可见性的问题，synchronized 却同时解决了互斥的问题（在该程序中不是个问题）。更重要的是，你还往程序中引进了另一个更严重的问题。

你已经正确地对 stopped 进行了同步访问，但是进一步观察 run()方法中的同步块，尤其是这个 while 循环。由于正在执行循环的这个线程已经获取了当前 StoppableThread 对象（通过 synchronized(this)）的锁，这个循环不会终止。因为默认的主线程需要获取相同的锁，所以它在该对象上调用 stopThread()方法的任意尝试都会导致自己被阻塞住。

你可以使用局部变量并在同步块中将 stopped 的值赋给这个变量来解决这一问题，如下所示：

```
public void run()
{
    boolean _stopped = false;
    while (!_stopped)
```

```
    {
        synchronized(this)
        {
            _stopped = stopped;
        }
        System.out.println("running");
    }
}
```

不过，每次循环迭代都要尝试获取锁的方式会存在性能开销（还不如以前），所以这个解决方式是得不偿失的。清单 2-4 展示了一个更为高效且整洁的方法。

清单 2-4　尝试通过 volatile 关键字来停止一个线程

```
public class ThreadStopping
{
    public static void main(String[] args)
    {
        class StoppableThread extends Thread
        {
            private volatile boolean stopped; // defaults to false

            @Override
            public void run()
            {
                while(!stopped)
                    System.out.println("running");
            }

            void stopThread()
```

```
        {
            stopped = true;
        }
    }
    StoppableThread thd = new StoppableThread();
    thd.start();
    try
    {
        Thread.sleep(1000); // sleep for 1 second
    }
    catch (InterruptedException ie)
    {
    }
    thd.stopThread();
  }
}
```

由于 stopped 已经标记为 volatile，每条线程都会访问主存中该变量的拷贝而不会访问缓存中的拷贝。这样，即使在多处理器或者多核的机器上，该程序也会停止。

警告：只有可见性导致问题时，才应该使用 volatile。而且，你也只能在属性声明处才能使用这个保留字（如果你尝试将局部变量声明成 volatitle，会收到一个错误）。最后，你可以将 double 和 long 型的属性声明成 volatile，但是应该避免在32位的JVM上这样做，原因是此时访问一个double或者long型的变量值需要进行两步操作，若要安全地访问它们的值，互斥（通过 synchronized）是必要的。

当一个属性变量声明成 volatile，就不能同时被声明 final 的。不过，由于 Java 可以让你安全地访问 final 的属性而无需同步，这也就不能称之为一个问题了。为了克服 DeadlockDemo 中的缓存变量问题，我把 lock1 和 lock2 都标记成 final，尽管也能将它们标记成 volatile 的。

以后，你会经常使用 final 关键字来确保在不可变（不会发生改变）类的上下文中线程的安全性。参考清单 2-5。

清单 2-5 借助于 final 创建一个不可变且线程安全的类

```
import java.util.Set;
import java.util.TreeSet;

public final class Planets
{
   private final Set<String> planets = new TreeSet<>();

   public Planets()
   {
      planets.add("Mercury");
      planets.add("Venus");
      planets.add("Earth");
      planets.add("Mars");
      planets.add("Jupiter");
      planets.add("Saturn");
      planets.add("Uranus");
      planets.add("Neptune");
   }

   public boolean isPlanet(String planetName)
```

```
    {
        return planets.contains(planetName);
    }
}
```

清单 2-5 展示了一个不可变类 Planets，其对象存储着星球名字的集合。尽管集合是可变的，但这个类的设计却保证在构造函数退出之后，集合不会再被改变。通过声明 planets 为 final，这个属性的引用不能被更改。而且，该引用也不能被缓存，所以缓存变量的问题也不复存在。

关于不可变对象，Java 提供了一种特殊的线程安全的保证。即便没有用同步来发布（暴露）这些对象的引用，它们依然可以被多条线程安全地访问。不可变对象提供了下列易于识别的规则：

- 不可变对象绝对不允许状态变更。
- 所有的属性必须声明成 final。
- 对象必须被恰当地构造出来以防 this 引用脱离构造函数。

最后一点很让人迷惑，所以这里给出一个 this 显式地脱离构造函数的简单例子：

```
public class ThisEscapeDemo
{
   private static ThisEscapeDemo lastCreatedInstance;

   public ThisEscapeDemo()
   {
      lastCreatedInstance = this;
   }
}
```

在 www.ibm.com/developerworks/library/j-jtp0618/上查看《Java theory and practice：Safe construction techniques》学习更多常见线程风险的相关知识。

2.5 练习

下面的练习用来测试你对第 2 章内容的掌握程度。

1. 指出与线程相关的三个问题。

2. 判断对错：当程序计算的正确性取决于相对时间或者调度器所控制的多线程交叉时，你会遇到数据竞争问题。

3. 给出同步的定义。

4. 指出同步的两种属性。

5. 同步是如何实现出来的？

6. 判断对错：当一个已经获取锁的线程调用任意 Thread 的任意 sleep() 方法时不会释放锁。

7. 如何指定一个同步方法？

8. 如何指定一个同步块？

9. 给出活跃性的定义。

10. 指出三种活跃性挑战。

11. 如何区分 volatile 和 synchronized 关键字？

12. 判断对错：Java 让你能够安全地访问一个 final 的属性而无需进行同步。

13. 指出下面 CheckingAccount 类存在的线程问题：

```
public class CheckingAccount
{
   private int balance;
   public CheckingAccount(int initialBalance)
   {
      balance = initialBalance;
   }
   public boolean withdraw(int amount)
   {
      if (amount <= balance)
      {
         try
         {
            Thread.sleep((int) (Math.random() * 200));
         }
         catch (InterruptedException ie)
         {
         }
         balance -= amount;
         return true;
      }
      return false;
   }
   public static void main(String[] args)
   {
      final CheckingAccount ca = new CheckingAccount(100);
      Runnable r = new Runnable()
                  {
```

```
            @Override
            public void run()
            {
               String name = Thread.currentThread().
               getName();
               for (int i = 0; i < 10; i++)
                  System.out.println (name + "
                  withdraws $10: " +
                                    ca.withdraw(10));
            }
         };
      Thread thdHusband = new Thread(r);
      thdHusband.setName("Husband");
      Thread thdWife = new Thread(r);
      thdWife.setName("Wife");
      thdHusband.start();
      thdWife.start();
   }
}
```

14. 修复前面 CheckingAccount 类的线程问题。

2.6 小结

线程交互通常是通过共享变量完成的，当线程之间没有交互，开发多线程的应用程序会变得简单许多。一旦发生了交互，竞态条件、数据竞争以及缓存变量等诱发线程不安全（在多线程环境下不正确）的因素就会暴露出来。

你可以使用同步解决之前的线程问题。同步是 JVM 的一个特性，旨在保证两条或者两条以上并发的线程不会同时进入同一块临界区。临界区就是必须以串行方式访问的一段代码块。

活跃性这个词代表着某件正确的事情最终会发生。活跃性失败发生在应用程序触及一种无法继续执行的状态。在单线程的应用程序中，无限循环就是一个例子。多线程应用程序面临着诸如死锁、活锁和饿死的额外挑战。

同步展示了两种属性：互斥性和可见性。`synchronized` 关键字与两者都有关系。Java 同时也提供了一种更弱的、仅仅包含可见性的同步形式，并且只以 `volatile` 关键字关联。

当一个属性变量声明成 `volatile`，就不能同时声明成 `final` 的。不过，由于 Java 可以让你安全地访问 `final` 的属性而无需同步，这也就不能称之为一个问题了。以后你会经常使用 `final` 来确保在不可变（不会发生改变）类的上下文中线程的安全性。

第 3 章会涉及等待和通知相关的话题。

第3章

等待和通知

Java 提供了一组很小的 API 来支持线程间交互。在使用这组 API 时，线程持续等待某个条件（继续执行的前置条件）成立，后续会有另一条线程创造出这一条件，进而通知处于等待中的线程。本章会介绍这组 API。

3.1 等待/通知 API 一览

`java.lang.Object` 类提供了一套等待/通知的 API，它由 3 个 `wait()`、一个 `notify()`和一个 `notifyAll()`方法组成。`wait()`方法等待某个条件成立，当这个条件成立时，`notify()`和 `notifyAll()`方法通知处于等待中的线程。

- `void wait()`：导致当前线程一直处于等待，直到另外的线程调用这个对象的 `notify()`或者 `notifyAll()`方法，又或者一直等待其他的线程中断当前等待的线程。
- `void wait(long timeout)`：导致当前线程一直处于等待，直到另外的线程调用这个对象的 `notify()`或者 `notifyAll()`方法，或者等待特定的毫秒数（由 `timeout` 确定），又或者一直等待到其他的线程

中断当前等待的线程。当 timeout 是负数的时候，这个方法抛出 java.lang.IllegalArgumentException。

- void wait(long timeout, int nanos)：导致当前线程一直处于等待，直到另外的线程调用这个对象的 notify()或者 notifyAll()方法，或者等待特定的毫秒（由 timeout 确定）和纳秒数（由 nanos 确定），又或者一直等待到其他的线程中断当前等待的线程。当 timeout 是负数、nanos 是负数，或者 nanos 大于 999999 的时候，这个方法抛出 java.lang.IllegalArgumentException。
- void notify()：唤醒正在等待该对象监听器的单条线程。如果有几条线程在该对象上等待，其中某一条会被挑选出来唤醒，这种选择是随意的且取决于具体实现。除非当前线程释放了该对象上的锁，被唤醒的线程仍不能执行。被唤醒的线程跟平常一样，同其他可能活跃的线程一起竞争在该对象上进行同步。举个例子，被唤醒的线程在接下来锁住对象这件事上，既没有优势也没有劣势。
- void notifyAll()：唤醒正在等待该对象监听器的全部线程。除非当前线程释放了该对象上的锁，否则，这些被唤醒的线程仍不能执行。被唤醒的线程跟平常一样，同其他可能活跃的线程一起竞争在该对象上进行同步。同上，被唤醒的线程在接下来锁住对象这件事上，既没有优势也没有劣势。

若当前线程开始或正在等待通知，任意线程中断了它，3 个 wait()方法都会抛出 java.lang.InterruptedException。此时，当前线程的中断状态会被清除。

注意：对象的 wait()方法被调用时，线程会放弃对象关联的监听器的所有权。

这组 API 利用了一个对象的条件队列，该队列就是一种数据结构，用于存储那些等待某个条件成立的线程。这些等待的线程称为等待集合。由于该条件队列和对象的锁紧紧捆绑在一起，以上 5 个方法必须在同步的上下文中（当前的线程必须是该对象监听器的所有者）被调用；否则，会抛出 java.lang.IllegalMonitorStateException。

下面的代码、伪代码片段演示了这个无参的 wait()方法：

```
synchronized(obj)
{
  while (<condition does not hold>)
    obj.wait();

  // Perform an action that's appropriate to condition.
}
```

wait()方法在一个同步块中调用，该同步块同步的对象和调用 wait()方法的对象是一样的。由于有假唤醒的可能性（线程不是因为通知、中断或者超时被唤醒），wait()方法在一个 while 循环中使用，这个循环会测试是否满足条件，并且当条件依然不满足时重新运行 wait()方法。while 循环退出之后，条件成立了，和条件相宜的动作会被执行。

警告： 绝对不要在循环外面调用 wait()方法。这个循环会在 wait()调用前后进行条件测试。在调用 wait()之前测试，确保了活跃性。如果这个测试不存在，但条件满足，同时 notify()先于 wait()方法调用，等待中的线程就不太可能再醒来。而调用 wait()之后重新测试条件，保证了安全性。如果没有重新测试，并且线程已经从 wait()中醒来（或许条件不满足时，其他线程意外地调用了 notify()方法），同时条件又没有满足，这个线程可能会继续破坏锁所维护的一致性。

下面的代码片段展示了 notify()方法，它通知了前面例子中等待的线程：

```
synchronized(obj)
{
   // Set the condition.

   obj.notify();
}
```

注意 notify()方法是从临界区中调用的，这个临界区和 wait()方法所在的临界区一样，被同一个对象（obj）所保护，并且也使用了同一个对象引用来调用 notify()方法。只要遵照这个模式，你就不会陷入麻烦。

注意：关于 notify()和 notifyAll()哪种通知方法更好，存在着大量争论。举个例子，参考《Difference between notify() and notifyAll()》（http://stackover flow.com/questions/14924610/difference-between-notify-and-notifyall）。如果你在犹豫用哪个方法，我会说，在应用程序中，仅有两条线程并且某条线程偶尔等待、需要被另一条线程通知的时候，才使用 notify()方法，否则使用 notifyAll()方法。

3.2 生产者和消费者

生产者和消费者线程之间的关系是涉及条件的线程交互的一个经典例子。生产者线程产生数据项，将来会被消费者消费。每个生产出来的数据项会存储在一个共享的变量中。

假设线程以不同的速度前行，生产者可能在消费者取得旧数据项处理之前生产了一个新数据项并且将其记录到共享变量中。当然，消费者也可能在新数据项生产出来之前就获取共享变量中的内容。

为了克服这些问题，生产者线程必须一直等待，直到它被通知之前生产的数据项已经被消费，并且消费者线程也必须一直等待，直到自己被通知新数据项已经被生产出来了。清单3-1展示了如何通过`wait()`和`notify()`方法完成这项任务。

清单3-1 生产者-消费者关系第1版

```
public class PC
{
   public static void main(String[] args)
   {
      Shared s = new Shared();
      new Producer(s).start();
      new Consumer(s).start();
   }
}

class Shared
{
   private char c;
   private volatile boolean writeable = true;

   synchronized void setSharedChar(char c)
   {
      while (!writeable)
         try
         {
```

```
                wait();
            }
            catch (InterruptedException ie)
            {
            }
        this.c = c;
        writeable = false;
        notify();
    }

    synchronized char getSharedChar()
    {
        while (writeable)
            try
            {
                wait();
            }
            catch (InterruptedException ie)
            {
            }
        writeable = true;
        notify();
        return c;
    }
}

class Producer extends Thread
{
```

```
    private final Shared s;

    Producer(Shared s)
    {
        this.s = s;
    }

    @Override
    public void run()
    {
        for (char ch = 'A'; ch <= 'Z'; ch++)
        {
            s.setSharedChar(ch);
            System.out.println(ch + " produced by producer.");
        }
    }
}
class Consumer extends Thread
{
    private final Shared s;

    Consumer(Shared s)
    {
        this.s = s;
    }

    @Override
    public void run()
    {
```

```
        char ch;
        do
        {
            ch = s.getSharedChar();
            System.out.println(ch + " consumed by consumer.");
        }
        while (ch != 'Z');
    }
}
```

这个应用程序创建了一个共享对象，并且两条线程各自获取了该对象的一份拷贝。生产者调用这个对象的 setSharedChar()方法以保存 26 个大写的英语字母；消费者则调用该对象的 getSharedChar()方法来获取每个字母。

writeable 实例变量跟踪了两个条件：生产者等待消费者消费一个数据项和消费者等待生产者生产一个新数据项。它帮助协调生产者和消费者的执行。下面以消费者先执行的场景来演示这种协调的做法：

（1）消费者执行 s.getSharedChar()方法以索取一个字母。

（2）在那个同步方法中，由于 writeable 变量是 true，所以消费者调用了 wait()方法。现在，消费者会一直等待，直到收到来自于生产者的通知。

（3）生产者最终执行了 s.setSharedChar(ch);方法。

（4）当生产者进入同步方法中（这是可能的，因为消费者在进入等待之前于 wait()方法中释放掉了锁），它发现 writeable 的值是 true，便不会调用 wait()方法。

（5）生产者保存下这个字母，把 writeable 的值设成 false（如果消费者到那时还没有消费这个字母，就会导致生产者在下次 setSharedChar()调用时陷入等待），并且调用 notify()方法去唤醒消费者（假设消费者还在等待）。

（6）生产者退出 setSharedChar(char c)方法。

（7）消费者醒来（同时重新获取锁），把 writeable 设成 true（如果生产者到那时还没能生产出一个字母，就会导致消费者在下次getSharedChar()调用时陷入等待），通知唤醒生产者线程（假设生产者还在等待），并返回共享的字母。

照下面的编译清单 3-1：

```
javac PC.java
```

运行程序：

```
java PC
```

你应该能在第一次运行的时候观察到如下的输出：

```
W produced by producer.
W consumed by consumer.
X produced by producer.
X consumed by consumer.
Y produced by producer.
Y consumed by consumer.
Z produced by producer.
Z consumed by consumer.
```

尽管这个同步正确地工作了，你可能还观察到，在多条消费信息前面还存在多条生产消息：

```
A produced by producer.
B produced by producer.
A consumed by consumer.
B consumed by consumer.
```

同时，你也可能观察到一条消费消息在一条生产信息前面：

```
V consumed by consumer.
V produced by producer.
```

奇怪的输出顺序并不意味着生产者和消费者线程就不是同步的。相反，这是紧跟着调用 setSharedChar()之后的伴随方法 System.out.println()没被同步，以及紧跟着调用 getSharedChar()之后的伴随方法 System.out.println()没被同步的结果。这个输出的顺序可以通过将每个方法调用对包装到一个同步块的方式修正，这个同步块同步在 s 所引用的 Shared 对象之上。清单 3-2 展示了这一改善。

清单 3-2　生产者-消费者关系第 2 版

```
public class PC
{
   public static void main(String[] args)
   {
      Shared s = new Shared();
      new Producer(s).start();
      new Consumer(s).start();
   }
}

class Shared
{
   private char c;
   private volatile boolean writeable = true;

   synchronized void setSharedChar(char c)
```

```
{
    while (!writeable)
        try
        {
            wait();
        }
        catch (InterruptedException ie)
        {
        }
    this.c = c;
    writeable = false;
    notify();
}

synchronized char getSharedChar()
{
    while (writeable)
        try
        {
            wait();
        }
        catch (InterruptedException ie)
        {
        }
    writeable = true;
    notify();
    return c;
}
```

```
}
class Producer extends Thread
{
   private final Shared s;

   Producer(Shared s)
   {
      this.s = s;
   }

   @Override
   public void run()
   {
      for (char ch = 'A'; ch <= 'Z'; ch++)
      {
         synchronized(s)
         {
            s.setSharedChar(ch);
            System.out.println(ch + " produced by producer.");
         }
      }
   }
}
class Consumer extends Thread
{
   private final Shared s;
   Consumer(Shared s)
   {
```

```
        this.s = s;
    }

    @Override
    public void run()
    {
        char ch;
        do
        {
            synchronized(s)
            {
                ch = s.getSharedChar();
                System.out.println(ch + " consumed by consumer.");
            }
        }
        while (ch != 'Z');
    }
}
```

编译清单 3-2（javac PC.java），然后运行程序（java PC）。它的输出应该总是以下面展示的交替顺序出现（为了简洁，这里只显示了前面的几行）。

```
A produced by producer.
A consumed by consumer.
B produced by producer.
B consumed by consumer.
C produced by producer.
C consumed by consumer.
D produced by producer.
D consumed by consumer.
```

3.3 练习

下面的练习用来验证你对第 3 章内容的掌握程度。

1. 给出条件的定义。

2. 描述支持条件的 API。

3. 判断对错：`wait()`方法可以被中断。

4. 你会调用什么方法来唤醒在同一对象监听器上等待的全部线程？

5. 判断对错：一条已经获取到锁的线程在调用 `Object` 的 `wait()`方法时并不会释放掉锁。

6. 给出条件队列的定义。

7. 当你在同步上下文之外调用这组 API 的方法时会发生什么？

8. 给出假唤醒的定义。

9. 为什么你需要在一个循环的上下文中调用 `wait()`方法？

10. 创建一个等待的应用程序，示范一种被称作关卡的高级别的同步构造。这一构造允许多条线程到达同一个同步点（关卡），并且一直等待直到关卡被其他的线程解锁，才能继续全部执行。

`main()`方法首先会为这些线程创建一个 `runnable`，线程会在关卡处等待。`runnable` 打印一条消息表明这条线程正处于等待中，同时递增一个计数器，睡眠 2 秒，然后等待（确保考虑了假唤醒问题）。唤醒后，线程输出一条消息，表明线程终止。之后 `main()`方法创建 3 个线程对象并且启动这 3 条线程去执行 runnable。接下来，`main()`方法创建另一个 `runnable`，它会反复地睡 200

毫秒直到计数器等于3，这时，它会通知所有处于等待中的线程。最后，main()方法为第二个runnable创建一个线程对象并启动之。

3.4 小结

Java提供了一组很小的API支持线程间交互。这组API由3个wait()、一个notify()和一个notifyAll()方法组成。wait()方法等待某个条件成立，当这个条件成立时，notify()和notifyAll()方法通知处于等待中的线程。

wait()、notify()和notifyAll()方法在一个同步块中被调用，该同步块同步的对象和调用wait()方法的对象是一样的。由于有假唤醒的可能性（线程不是因为通知、中断或者超时被唤醒），wait()方法在一个while循环中使用，这个循环会测试是否满足条件，并且当条件依然不满足时，重新运行wait()方法。

生产者和消费者线程之间的关系是涉及条件的线程交互的一个经典例子。生产者线程产生数据项，将来会被消费者消费。每个生产出来的数据项会被存储在一个共享的变量中。

为了克服那些诸如数据项还没被生产出来就被消费的问题，生产者线程必须一直等待，直到它被通知之前生产的数据项已经被消费，并且消费者线程也必须一直等待，直到自己被通知新数据项已经被生产出来了。

第4章会涉及额外的线程能力。

第4章

额外的线程能力

第1章到第3章介绍了java.lang.Thread类、java.lang.Runnable接口、同步、等待以及通知等内容。在本章中，我会以线程组和线程局部变量来结束有关线程基础知识的讨论。同时，我也会介绍定时器框架以及其如何利用幕后的线程简化面向定时器的任务。

4.1 线程组

查看Thread类的时候，你可能会在构造函数中看到java.lang.ThreadGroup的引用，如Thread(ThreadGroup group,Runnable target)，又或者在static int activeCount()和static int enumerate(Thread[] tarray)方法文档中看到相关的描述。

对于ThreadGroup，JDK文档这样阐述：一个线程组代表了一组线程。除此之外，一个线程组也能包含其他的线程组。这些线程组形成一棵树，其中除了初始线程，每个线程组都有一个父线程组。

使用 ThreadGroup 对象，你可以对其中的所有线程对象进行操作。举个例子，假设一个线程组被变量 tg 引用，tg.suspend();会暂停线程组内的所有线程。线程组简化了多条线程的管理工作。

尽管线程组看上去很有用，但是有碍于下列原因，你万万不能使用这个类：

- ThreadGroup 中最有用的方法就是 void suspend()、void resume()以及 void stop()，然而这些方法都已经被废弃了，原因和 Thread 类似（这些方法分派给该线程组的各个线程），它们很容易诱发死锁等问题。
- ThreadGroup 是非线程安全的。举个例子，为了获取线程组中的活跃线程数，你会调用 ThreadGroup 的 int activeCount()方法。然后使用此方法的返回值限定传递给 ThreadGroup 的某个 enumerate()方法的数组的大小。不过，这并不会保证线程数是精确的，原因在于数组创建完成到被传递给 enumerate()方法期间，存在线程创建与消亡带来的数量改变。如果数组太小了，enumerate()会悄悄地忽略掉多余的线程。同样的情况也适用于 Thread 类的 activeCount()和 enumerate()方法，它们会被代理到当前线程所在 ThreadGroup 的方法上。这就是一个“检查时间到使用时间”（https://en.wikipedia.org/wiki/Time_of_check_to_time_of_use）类别的软件缺陷（进行操作之前需检查文件是否存在的场景中，这个缺陷也会露出“爪牙”。因为检查文件和操作期间，文件也可能被删除或创建）。

不过，由于 ThreadGroup 在处理线程执行过程中产生的异常方面做出的贡献，你还是应该对它有所了解。清单 4-1 通过一个尝试用整数除以 0 导致抛出 java.lang.ArithmeticException 的 run()方法来示范了异常处理。

清单 4-1 从 run()方法中抛出一个异常

```
public class ExceptionThread
{
   public static void main(String[] args)
   {
      Runnable r = new Runnable()
                   {
                      @Override
                      public void run()
                      {
                         int x = 1 / 0; // Line 10
                      }
                   };
      Thread thd = new Thread(r);
      thd.start();
   }
}
```

默认主线程创建了一个尝试除 0 而故意抛出 ArithmeticException 对象的 runnable。

按照下面的方式编译清单 4-1：

```
javac ExceptionThread.java
```

运行程序：

```
java ExceptionThread
```

你会看到类 ArithmeticException 的实例被抛出的一条异常栈信息：

```
Exception in thread "Thread-0" java.lang.ArithmeticException: / by zero
        at ExceptionThread$1.run(ExceptionThread.java:10)
        at java.lang.Thread.run(Thread.java:745)
```

一旦run()方法抛出异常，线程就会中止并被下列活动取代：

- Java 虚拟机（JVM）寻找 Thread.UncaughtExceptionHandler 的实例，该实例由 Thread 的 void setUncaughtExceptionHandler (Thread.UncaughtExceptionHandler eh)方法设置。当找到这个 handler 时，线程就会执行它的 void uncaughtException(Thread t, Throwable e)方法，这里 t 代表了抛出异常线程所关联的 Thread 对象，而 e 代表了被抛出的异常或者错误本身——可能会抛出一个 java.lang.OutOfMemoryError 对象。如果 uncaughtException() 抛出了一个异常或错误，这个异常或错误会被 JVM 忽略。
- 假设 setUncaughtExceptionHandler()未经调用，JVM 就把控制权交给线程关联的 ThreadGroup 对象上的 uncaughtException (Thread t,Throwable e) 方法。假设这个 ThreadGroup 没被继承且 uncaughtException()方法没被重写，那么此时若其父 ThreadGroup 存在，uncaughtException()又会把控制权移交给其父 ThreadGroup 的 uncaughtException()方法。否则，就检查默认的未捕获异常处理器是否已经被设置（通过 Thread 的 static void setDefault UncaughtExceptionHandler(Thread.UncaughtExceptionHandler handler)方法）。如果默认未捕获异常处理器已经被设置过，其 uncaughtException()方法就会被传入相同的两个参数进行调用。反之，uncaughtException()方法检查其 Throwable 参数是否为 java.lang.ThreadDeath 的实例。如果是，就没什么特别的了。否则，

就像清单 4-1 中异常消息显示的那样，一条包含了线程名称和错误栈的消息就被打印到标准错误流中，线程名称返回自该线程的 getname()方法，错误栈出自 Throwable 栈的 printStackTrace()的方法。

清单 4-2 演示了 Thread 的 setUncaughtExceptionHandler()和 setDefaultUncaughtExceptionHandler()方法。

清单 4-2 未捕获异常处理器示例

```
public class ExceptionThread
{
   public static void main(String[] args)
   {
      Runnable r = new Runnable()
                   {
                      @Override
                      public void run()
                      {
                         int x = 1 / 0;
                      }
                   };
      Thread thd = new Thread(r);
      Thread.UncaughtExceptionHandler uceh;
      uceh = new Thread.UncaughtExceptionHandler()
            {
               @Override
               public void uncaughtException(Thread t, Throwable e)
               {
                  System.out.println("Caught throwable " + e +
```

```
                                        " for thread " + t);
                }
            };
        thd.setUncaughtExceptionHandler(uceh);
        uceh = new Thread.UncaughtExceptionHandler()
                {
                    @Override
                    public void uncaughtException(Thread t, Throwable e)
                    {
                        System.out.println("Default uncaught exception handler");
                        System.out.println("Caught throwable " + e +
                                           " for thread " + t);
                    }
                };
        thd.setDefaultUncaughtExceptionHandler(uceh);
        thd.start();
    }
}
```

编译清单 4-2（javac ExceptionThread.java），并运行最终程序（java ExceptionThread）。你应该能观测到这样的输出：

```
Caught throwable java.lang.ArithmeticException: / by zero for thread
Thread[Thread-0,5,main]
```

因为默认的处理器并未被调用，你看不到它的输出。想要看到那样的输出，你必须注释掉 thd.setUncaughtExceptionHandler(uceh);这行。如果你也注释掉了 thd.setUncaughtExceptionHandler(uceh);这行，那么就会看到原本清单 4-1 中的输出。

4.2 线程局部变量

有时候，你想要把线程私有数据（如一个用户 ID）和线程关联起来。尽管你可以用一个局部变量来完成这项任务，那也得它确实存在才行。你可能会使用一个实例属性来保存数据，但是这样你又不得不处理同步问题了。好在，Java 提供了类 `java.lang.ThreadLocal` 作为一个简单（还非常好用）的可选方案。

每个 `ThreadLocal` 的实例代表了一个线程局部变量，它为每一条访问线程提供了单独的存储槽（storage slot）。你可以把它想象成具有多个槽的变量，然后每条线程可以在同一个变量中存储不同的值。并且，每条线程都只能看到自己的值，而不会意识到其他线程在这个变量中也有自己的值。

`ThreadLocal` 被声明成泛型的 `ThreadLocal<T>`，这里的 `T` 标识了存储在该变量中值的类型。这个类声明了如下的构造函数和方法：

- `ThreadLocal()`：创建了一个新的线程局部变量。
- `T get()`：返回调用线程的存储槽中的值。如果当这个线程调用此方法时，值不存在，那么 `get()`会调用 `initialValue()`方法。
- `T initialValue()`：创建调用线程的存储槽并存入一个初始（默认）值。默认的初始值是 `null`。你必须继承类 `ThreadLocal` 并重写其 `protected` 方法以提供一个更加合适的初始值。
- `void remove()`：清空调用线程的存储槽。在没有 `set()`方法介入的情况下，如果紧随此方法之后调用 `get()`方法，那么 `get()`就会直接调用 `initialValue()`。

- `void set(T value)`：设置调用线程存储槽上的值。

清单 4-3 展示了如何使用 `ThreadLocal` 把不同的用户 ID 关联到两条线程上。

清单 4-3　为不同的线程关联不同的用户 ID

```
public class ThreadLocalDemo
{
   private static volatile ThreadLocal<String> userID =
      new ThreadLocal<String>();

   public static void main(String[] args)
   {
      Runnable r = new Runnable()
                   {
                      @Override
                      public void run()
                      {
                         String name = Thread.currentThread().getName();
                         if (name.equals("A"))
                            userID.set("foxtrot");
                         else
                            userID.set("charlie");
                         System.out.println(name + " " + userID.get());
                      }
                   };
      Thread thdA = new Thread(r);
      thdA.setName("A");
      Thread thdB = new Thread(r);
```

```
        thdB.setName("B");
        thdA.start();
        thdB.start();
    }
}
```

初始化 ThreadLocal 并将引用赋值给一个 volatile 的类属性 userID（由于这个属性可能会运行在多处理器/多核的机器上，被多条线程访问，所以它是 volatile 的，也可以指定 final 来替代）之后，默认的主线程创建了两条线程，这两条线程分别在 userID 中存储了不同的 java.lang.String 对象并随后打印出这些对象。

照下面这样编译清单 4-3：

```
javac ThreadLocalDemo.java
```

运行程序：

```
java ThreadLocalDemo
```

你应该观察到如下的输出（顺序可能有所不同）：

```
A foxtrot
B charlie
```

存储在线程本地变量中的值都是不相关的。当一个新的线程被创建出来，它就会获得一个新的包含 initialValue()值的存储槽。有时候你或许想要把值从父线程（一个创建其他线程的线程）传给子线程（被创建的线程），那么借助 InheritableThreadLocal 就可以完成这项任务。

InheritableThreadLocal 是 ThreadLocal 的子类。除了定义了一个 InheritableThreadLocal()构造方法，这个类还声明了下面的 protected 方法：

- `T childValue(T parentValue)`：在子线程被创建出来的时候，用父线程的值（作为参数）计算出子线程的初始值。此方法会在子线程启动之前被父线程调用。若它没被重写就会返回参数 `parentValue`。反之，返回期望的值。

清单 4-4 展示了如何使用 `InheritableThreadLocal` 将父线程中的 Integer 对象传给子线程中。

清单 4-4　将一个对象从父线程传到子线程中

```
public class InheritableThreadLocalDemo
{
   private static final InheritableThreadLocal<Integer> intVal =
      new InheritableThreadLocal<Integer>();

   public static void main(String[] args)
   {
      Runnable rP = () ->
                    {
                       intVal.set(new Integer(10));
                       Runnable rC = () ->
                                     {
                                        Thread thd = Thread.currentThread();
                                        String name = thd.getName();
                                        System.out.printf("%s %d%n", name,
                                                          intVal.get());
                                     };
                       Thread thdChild = new Thread(rC);
                       thdChild.setName("Child");
                       thdChild.start();
                    };
```

```
        new Thread(rP).start();
    }
}
```

初始化 InheritableThreadLocal 之后，将其赋值给一个用 final（也可以用 volatile 替代）修饰的、名为 intVal 的类属性，默认主线程创建了一条父线程，这条线程在 intVal 中存储了一个值为 10 的 java.lang.Integer 对象。父线程之后创建了一条子线程，这条线程访问 intVal 并取得父线程中的 Integer 对象。

编译清单 4-4：

```
javac InheritableThreadLocalDemo.java
```

运行程序：

```
java InheritableThreadLocalDemo
```

你应该能够看到如下输出：

```
Child 10
```

注意：更多关于 ThreadLocal 的洞见以及实现方式，请查看 Patson Luk 的《A Painless Introduction Java ThreadLocal Storage》博客文章（http://java.dzone.com/articles/painless-introduction-javas-threadlocal-storage）。

4.3 定时器框架

通常情况下，安排单次的执行任务或者规律性的重复任务是有必要的。举个例子，你很可能定制了只运行一次的闹钟任务（或许是为了早上叫你起床）

或者一个夜间定时的备份任务。不论是哪种类型的任务，你可能都需要它在未来某个指定的时间点或者经过一段初始延时后执行。

你可以使用线程和相关类型构造一个任务调度的框架。不过，Java 1.3 已经以 **java.util.Timer** 和 **java.util.TimerTask** 的形式引入了一个更加便利和简单的替代方案。

Timer 让你能够在一个后台线程中调度 **TimerTasks** 用于后续执行（以顺序的方式），它也称为任务执行线程。定时器任务可能会因为单次的执行或者规律性的重复执行而被调度。

清单 4-5 展示了单次执行的定时器任务

清单 4-5 单次执行任务的示例

```
import java.util.Timer;
import java.util.TimerTask;

public class TimerDemo
{
   public static void main(String[] args)
   {
         TimerTask task = new TimerTask()
                        {
                           @Override
                           public void run()
                           {
                              System.out.println("alarm going off");
                              System.exit(0);
                           }
                        };
      Timer timer = new Timer();
```

```
        timer.schedule(task, 2000); // Execute one-shot timer task after
                                    // 2-second delay.

    }
}
```

清单 4-5 描述了这样一个应用程序，默认主线程首先初始化了一个 TimerTask 的匿名子类，由于程序要等到非守护任务执行线程终止之后才能终止，所以这个子类被重写的 run()方法会在输出一条警告信息后执行 System.exit(0);[1] 然后默认主线程初始化了 Timer，然后用 task 作为第 1 个参数调用 Timer 的 schedule()方法，此方法的第 2 个参数使得任务在经过 2000 毫秒的初始延时之后执行一次。

照下面的样子编译清单 4-5：

```
javac TimerDemo.java
```

运行程序：

```
java TimerDemo
```

你应该能够观测到类似下面的输出：

```
alarm going off
```

清单 4-6 展示了以固定时间间隔重复执行的一个定时器任务。

清单 4-6 以大约 1 秒的间隔不断显示当前系统时间

```
import java.util.Timer;
import java.util.TimerTask;


public class TimerDemo
```

[1] 译者注：终止程序。

```
{
    public static void main(String[] args)
    {
        TimerTask task = new TimerTask()
                        {
                            @Override
                            public void run()
                            {
                                System.out.println(System.currentTimeMillis());
                            }
                        };
        Timer timer = new Timer();
        timer.schedule(task, 0, 1000);
    }
}
```

清单 4-6 描述了这样一个应用程序，其默认主线程首先初始化了一个 TimerTask 的匿名子类，该子类被重写的 run()方法输出当前的系统时间（以毫秒的形式）。之后默认主线程初始化了 Timer 并将 task 作为第一个参数来调用 schedule()方法。方法的第 2 和第 3 个参数使得这个任务立即启动并且每隔 1000 毫秒执行一次。

编译清单 4-6（javac TimerDemo.java）并运行最终程序（java TimerDemo）。你应该能观测到如下的输出（这里进行了省略）：

```
1445655847902
1445655848902
1445655849902
1445655850902
```

```
1445655851902
1445655852902
```

4.3.1 深入 Timer

前面的应用程序以一种非守护任务执行线程的方式运行了它们自己的任务。当然，其中一个任务只执行一次，而另一个任务则重复执行。为了理解为何做出这些选择的，你需要更进一步学习 `Timer`。

注意：`Timer` 适用于大规模并发调度定时任务（数以千计的任务不在话下）。在内部，该类使用了一个二进制的堆表示其定时任务队列，使得调度定时任务的开销为 O(log n)，这里的 n 就是并发调度定时任务的数量。进一步了解 O()表示法，请查看维基页上“Big O notation”的主题（`http://en.wikipedia.org/wiki/Big_O_notation`）。

`Timer` 声明了下列构造函数。

- `Timer()`：创建一个新的定时器，其任务执行线程是非守护线程。
- `Timer(boolean isDaemon)`：创建一个新的定时器，其任务执行线程可能会以守护线程（传递 `true` 给 `isDaemon` 变量）的方式运行。守护线程会在定时器被用于调度重复的“维护活动”的场景下调用，只要应用程序处于运行状态，这些维护活动就会一直被执行，但是它们不会延长应用的存在时长。
- `Timer(String name)`：创建一个新的定时器，其任务执行线程拥有指定的名称且不会以守护线程的方式执行。当名称为 `null` 时，该构造函数就会抛出 `java.lang.NullPointerException`。

- `Timer(String name, boolean isDaemon)`：创建一个新的定时器，其任务执行线程拥有指定的名称且可能会以守护线程的方式执行。当名称为`null`时，这个构造函数就会抛出`java.lang.NullPointerException`。

`Timer`还声明了下列的方法。

- `void cancel()`：终止该定时器，丢弃所有当前调度的定时器任务。这个方法不会干涉当前正在执行的定时器任务（如果它存在的话）。定时器被终止之后，其执行线程会优雅地终止掉并且不会再有定时器任务在上面被调度（从这个定时器调用的定时任务的`run()`方法中调用`cancel()`方法，就能保证正在执行的任务一定是最后一个被该定时器执行的任务）。这个方法可能会被重复地调用，但第二次及后续的调用不会产生影响。
- `int purge()`：从该定时器队列中移除所有取消的定时任务并且返回被移除任务的数目。调用`purge()`方法不会对该定时器的行为产生影响，但是会从队列中删除被取消任务的引用。当这些定时器任务没有外部引用的时候，它们就可以被垃圾回收了[大部分应用程序不需要调用这个方法，该方法设计用于极少数需取消大量定时器任务的应用程序。调用`purge()`方法是在用时间换空间：此方法运行时（复杂度）可能同n+c * log(n)成比例，这里的n就是定时任务的总数，而c则是被取消任务的数目]。从这个定时器调度的定时任务中调用`purge()`方式是可行的。
- `void schedule(TimerTask task, Date time)`：在某个时间点调度任务执行。当`time`是过去时间，任务就会立即执行。当`time.getTime()`返回负数时，此方法抛出`java.lang.IllegalArgumentException`；若任务已经被调度或者取消、定时器被取消或者任务执行线程中断了，

则抛出 java.langIllegalStateException；如果 task 或者 time 是 null，则抛出 NullPointerException。

- void schedule(TimerTask task, Date firstTime, long period)：调度任务于 firstTime 开始，以固定时间间隔的方式重复执行，后续将以大约 period 毫秒数的固定时间间隔执行。在固定延迟执行中，每次执行都是相对于上次执行的实际发生时间的。当某次执行因为一些原因被延迟了（比如垃圾回收），后续的执行也都会被延迟。从长远来看，执行频率大体上会稍慢于对应的 period [假设在 Object.wait(long)下的系统时钟是精确的]。总而言之，当被调度的 firstTime 是过去时间，任务就会被立即执行。固定延时的执行对于需要“平滑”执行的重复任务是合适的。换句话说，这种类型的执行方式对于某些场景下的任务是合适的，即在短期内需要保持精准的频率。这就包括了大部分的动画任务，比如固定时间间隔的光标闪烁。也包括响应用户输入的规律性活动，比如只要一个键被按下就会自动出现一个字符。当 firstTime.getTime()返回负数或者 period 为非正数的时候，该方法抛出 IllegalArgument Exception；当任务已经被调度或取消、定时器被取消又或者任务执行线程终止时，抛出 IllegalStateException；若 task 或 firstTime 为 null，则抛出 NullPointerException。
- void schedule(TimerTask task,long delay)：在 delay 毫秒数之后调度任务执行。当 delay 为负数或者 delay + System. CurrentTime Millis()是负数时，该方法抛出 IllegalArgumentException；若任务已经被调度或者取消、定时器被取消或者任务执行线程终止，都会抛出 IllegalStateException；若 task 是 null，则抛出 NullPointerException。

- void schedule(TimerTask task,long delay,long period)：调度任务于 firstTime 开始，以固定时间间隔的方式重复执行，后续将以大约 period 毫秒数的固定时间间隔执行。当 delay 是负数，delay + System.currentTimeMillis()是负数的时候，或者 period 是非正数，该方法抛出 IllegalArgumentException；当任务已经被调度或取消、定时器被取消又或者任务执行线程终止时，都会抛出 IllegalStateException；若 task 或 firstTime 为 null，则抛出 NullPointException。
- void scheduleAtFixedRate(TimerTask task, Date firstTime, long period)：调度任务于 firstTime 开始，以固定速率的方式重复执行，后续将以大约 period 毫秒数的固定时间间隔执行。在以固定速率执行的方式当中，每次执行都是相对于初次执行的调度时间而被调度的。当某个执行因为一些原因被延迟了（比如垃圾回收），两次及两次以上的执行将会迅速相继发生。从长远来看，执行频率将会精确对应 period 的倒数［假设在 Object.wait(long)下的系统时钟是精确的］。总而言之，当被调度的 firstTime 是过去时间，任何“错过”的执行都会被调度立即“赶上来”执行。固定速率的执行方式适用于递归的活动，这些活动对绝对时间很敏感（比如每小时响一次的时钟或者每天固定时间点运行的调度维护任务）。它也适用于那些看重总时间并且执行次数固定的重复任务，比如 10 秒之中每秒滴答一次的倒计时器。最后，固定速率的执行方式对于调度多个重复的定时器任务也是适用的，这些任务彼此之间必须保持同步。当 firstTime.getTime()返回负数或者 period 为非正数的时候，该方法抛出 IllegalArgumentException；当任务已经被调度或取消、定时器被取消又或者任务执行线程终止时，抛出 IllegalStateException；若 task 或 firstTime 为 null，则抛出 NullPointerException。

- void scheduleAtFixedRate(TimerTask task, long delay, long period)：在 delay 毫秒数之后，开始调度任务以固定速率的方式重复执行，后续将以大约 period 毫秒数的固定时间间隔执行。当 delay 是负数，delay + System.currentTimeMills 为负数或者 period 为非正数的时候，该方法抛出 IllegalArgumentException；当任务已经被调度或取消、定时器被取消又或者任务执行线程终止时，抛出 IllegalStateException；若 task 为 null，则抛出 NullPointerException。

当最后一条指向 Timer 对象的引用消失并且所有重要的定时任务完成执行之后，该定时器的任务执行线程会优雅地终止（同时被垃圾回收）。不过，这个过程会在不定的时长里发生（默认情况下，这个任务执行线程不会以守护线程的方式运行，所以它能够阻止应用程序终止）。

当应用程序想要快速地终止一个定时器执行线程，它应该调用 Timer 的 cancel()方法。如果这个定时器的任务执行线程异常终止，举个例子，由于其 stop()方法被调用过（你绝不应该调用任何线程的 stop()方法，因为它们本身就是不安全的），以后针对该定时器尝试调度定时任务，都会导致抛出 IllegalStateException，就如同 Timer 的 cancel()方法已经被调用过一样。

4.3.2 深入 TimerTask

定时任务都是抽象类 TimerTask 子类的实例，这些子类实现了 Runnable 接口。当子类化 TimerTask 的时候，你需要重写其 void run()方法。

注意：定时任务应该很快完成。如果一个定时任务花费了很长的时间完成，它就会“霸占”定时器的任务执行线程，推迟后续定时任务的执行，这些后续任务可能会“裹成一团”，并且当这个侵入的定时任务最终完成，它们会接连快速地执行。

你也可以从定时任务被重写的run()方法中调用下列方法。

- boolean cancel()：取消这个定时任务。当该定时任务已经被调度成单次执行却还没有运行，又或者还没有被调度时，它就绝不会运行了。当该定时任务已经被调度成重复执行时，它就再不会执行了（若这个定时任务正在运行中，进行了此调用，那么该定时任务会完成运行，但是不会再次运行）。在重复型定时任务的run()方法中调用cancel()方法，一定能保证此定时任务不会再次运行。这个方法可能会被重复调用，第二次及其后续的调用不会产生影响。当定时任务被调度成单次执行且还没有运行，又或者被调度成重复执行时，此方法返回true。若定时任务被调度成单次执行且已经运行，或者从未被调度过，又或者已经被取消了，此方法会返回false（一般来说，当此方法阻止了一个或者多个被调度任务的执行时，它会返回true）。
- long scheduledExecutionTime()：返回此定时任务最近实际被调度执行的时间（当定时任务正在执行时此方法被调用了，返回值就是运行中定时任务的调度执行时间）。此方法通常会在一个任务的run()方法中被调用，以确定当前定时任务的执行是否足够适时地保证调度活动的运行。举个例子，你可能会在run()方法的开始指定像这样的代码if (System.currentTimeMillis() -scheduledExecutionTime() >= MAX_TARDINESS) return;如果当前定时任务不是适时的，就终止它。该方法通常不会和固定延迟执行、重复的定时任务一起使用，由于它们的调度执行时间允许随时间变化，因此不是特别得显著。scheduledExecutionTime()方法会返回被调度定时任务最近的执行时间，时间会以java.util.Date.getTime()的格式返回。若定时任务还没有开始首次执行，其返回值将是未定义的。

4.4 练习

下面的练习用来验证你对第 4 章内容的掌握程度。

1. 给出线程组的定义。

2. 你为何要使用线程组？

3. 你为何应该避免使用线程组？

4. 你为何应该认识线程组？

5. 给出线程局部变量的定义。

6. 判断对错：如果线程调用 `get()`方法时，调用线程的存储槽里没有值，该方法就会去调用 `initialValue()`。

7. 你如何从父线程传递一个值到子线程中？

8. 指出组成 `Timer` 框架的所有类。

9. 判断对错：`Timer()`会创建一个新的定时器，其任务执行线程会以守护线程的方式运行。

10. 给出固定延迟的执行任务的定义。

11. 你会调用哪些方法来调度一个固定延迟的执行任务？

12. 给出固定速率的执行任务的定义。

13. `Timer` 的 `cancel()`方法和 `TimerTask` 的 `cancel()`方法有何区别？

14. 创建一个 `BackAndForth` 应用程序，使用 `Timer` 和 `TimerTask` 去反复地移动一个星号：向前 20 步，然后向后 20 步。星号通过 `System.out.print()` 输出。

4.5　小结

类 ThreadGroup 描述了一个线程组，它存储了一组线程。线程组通过把方法应用到其中的所有线程而简化了多条线程的管理工作。由于多数有用的方法因竞态条件遭到废弃，你不应该再使用线程组。

类 ThreadLocal 描述了一个线程局部变量，它可以让你把每条线程的数据（如用户 ID）和线程关联。线程局部变量针对每个访问该变量的线程都提供了单独的存储槽。你可以把它想象成具有多个槽的变量，然后每条线程可以在这个变量中存储不同的值。每条线程只能看到自己的值而不会意识到其他的线程在这个变量中也有属于自己的值。存储在线程局部变量中的值都是不相关的。父线程可以借助类 InheritableThreadLocal 把值传递到子线程中。

通常情况下，安排单次的执行任务或者规律性的重复任务是有必要的。Java 1.3 引入了定时器框架，用于在定时器上下文中控制线程的执行，其由类 Timer 和 TimerTask 构成。

第 5 章会引入并发工具并展示 executors。

第 2 部分
并发工具类

■■■

第5章

并发工具类和Executor框架

前4章主要关注Java对底层线程操作的支持。这章开始关注Java对高级线程操作的支持，即并发工具类。你可以认为使用并发工具类和使用高级语言编写应用程序类似，而使用较底层的线程操作则和使用汇编语言编写应用程序类似。简单介绍完这些实用工具之后，我会带你浏览一遍Executor。后3章会涵盖其他多种并发工具类。

5.1 并发工具类简介

Java对底层线程操作的支持使得你可以创建多线程应用程序。这些程序比对应的单线程程序提供了更好的性能和响应能力。尽管如此，依然存在一些问题：

- 低级的并发原语，比如`synchronized`和`wait()`、`notify()`经常难以正确使用。对这些原语的误用会导致竞态条件、线程饿死、死锁和其他风险，这些风险都很难侦测和调试。
- 太过依赖`synchronized`原语会带来性能问题，这些问题会影响应用程序的可扩展性，对于诸如Web服务器之类的高度线程化的应用程序而言，后果会很严重。

- 开发者经常需要高级的线程结构，如线程池和信号量。Java 对底层线程操作的支持中不包含这些结构，开发者被迫构建一套自己的结构，这样不仅耗时还容易出错。

为了解决这些问题，Java 5 引入了并发工具类，这是一个由强大且易扩展的高性能线程工具类组成的框架，包含了线程池和阻塞队列。该框架由下列包中的多种类型组成。

- `java.util.concurrent`：经常在并发编程当中使用的工具类，如：`executors`。
- `java.util.concurrent.atomic`：支持在单个变量上进行无锁且线程安全操作的工具类。
- `java.util.concurrent.locks`：在某些条件上（这类对象会暂停执行“等待”线程直到被其他线程唤醒，此时对象的一些布尔型状态可能变为 `true`）获取锁和执行等待的工具类。通过这些类型进行锁或等待操作比通过 Java 基于监听器的同步以及等待、唤醒机制具有更好的性能和伸缩性。

这个框架也将 `long nanoTime()`方法引入了类 `java.lang.System` 中，这让你做相对时间测量的时候可以访问纳秒级别的时间资源。

并发工具类可以被分类为 `executor`、同步器（synchronizer）以及锁框架等。我会在接下来的章节探讨 `executor` 并在后续章节讨论其余类别。

5.2 探索 Executor

线程 API 使得你可以通过诸如 `new java.lang.Thread(new RunnableTask()).start();`的方式执行任务。这些表达式将多个任务的提交及它们的执行机制（在

当前线程、新线程或者随机地从线程池“组”中选择的线程）紧紧地绑在一起。

注意：一个任务是这样一种对象，它的类实现了java.lang.Runnable（一个可运行的任务）或者java.util.concurrent.Callable接口（一个可被调用的任务）。稍后我会介绍更多有关于Callable的内容。

并发工具类使用高级的executor替代了底层的线程操作执行任务。一个executor就是这样一种对象，它的类直接或间接地实现了java.util.concurrent.Executor接口，这样便从任务执行机制中解耦任务的提交操作。

注意：Executor框架使用接口从任务执行中解耦任务的提交操作，这和Collections框架使用接口从它们的实现中解耦lists、sets、queues以及maps类似。解耦会产生更易维护的、具备伸缩性的代码。

Executor声明了一个单独的void execute(Runnable runnable)方法，该方法会在将来的某个时间点执行这个名为runnable的可运行任务。当runnable是null时，execute()方法会抛出java.lang.NullPointerException；当无法执行runnable时，会抛出java.util.concurrent.RejectedExecutionException。

注意：当一个executor正在关闭并且不想接受新的任务时，RejectedExecutionException会被抛出来。当然，当executor没有足够的空间存储这一任务（或许executor使用有限的阻塞队列来存储任务且队列已满，我会在第8章讨论阻塞队列）时，这种异常也会被抛出来。

下面的例子展示了前面提及的new Thread(new RunnableTask()).start();表达式在Executor中的等价形式：

```
Executor executor = ...; //  ... represents some executor creation
executor.execute(new RunnableTask());
```

尽管 Executor 很好用，但这个接口还是在很多方面有所限制：

- Executor 只关注 Runnable 接口。由于 Runnable 的 run()方法没有返回值，这对于一个可运行的任务而言，就没法简单地把值返回给调用者。
- Executor 没有提供一种方式来追踪可运行任务的运行过程，例如，正在执行、正在取消了一个执行中的任务或者确认什么时候任务完成执行。
- Executor 无法执行一组可运行任务。
- Executor 没有为应用程序提供一种方式来关闭一个 executor（更为正确地关闭 executor）。

这些限制被 java.util.concurrent.ExecutorService 接口解决了，这个接口扩展 Executor 接口并且其实现就是一个典型的线程池。表 5-1 描述了 ExecutorService 的方法。

表 5-1　　ExecutorService 的方法

方法	说明
boolean awaitTermination (long timeout, TimeUnit unit)	在一条关闭请求之后，不论是任务全部都已完成、timeout（以 unit 的时间单元衡量）超时还是当前线程被中断的哪一条先发生，都会一直阻塞（等待）。如果 executor 已经终止，返回 true；若在终止之前超时，返回 false
<T> List<Future<T>> invokeAll (Collection<?extends Callable<T>>tasks)	执行任务集合中每一个 callable 任务并当所有的任务执行完成后（即任务要么正常结束，要么抛出异常），返回一个 java.util.List 的 java.util.concurrent.Future 的实例（本章稍后讨论），这些实例会持有任务的状态和结果。这组 Future 和任务迭代器返回的任务序列具有相同的顺序。当任务处于等待状态却被中断时，该方法抛出 InterruptedException，同时未完成的任务也会被取消；若 tasks 或者其任意元素为 null，抛出 NullPointerException；当没有任何任务可以被调度执行时，会抛出 RejectedExecutionException

续表

方法	说明
<T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks, long timeout, TimeUnit unit)	执行任务集合中每一个 callable 任务并当所有的任务执行完成后（即任务要么正常结束，要么抛出异常），或者 timeout（以 unit 的时间单元衡量）超时后，返回一组 java.util.concurrent.Future 的实例，这些实例会持有任务的状态和结果。超时未完成的任务都会被取消。这组 Futures 和任务迭代器返回的任务序列具有相同的顺序。当任务处于等待（未完成的任务会被取消）却被中断时，该方法抛出 InterruptedException。若 tasks 和其任意元素或 unit 为 null，抛出 NullPointerException；当没有任何任务可以被调度执行时，会抛出 RejectedExecutionException
<T> T invokeAny(Collection<? extends Callable<T>> tasks)	执行给出的 tasks，如果有任务成功完成执行（换句话说，没有抛出异常）就会返回其结果。只要有正常或者异常的返回，未完成执行的任务都会被取消。当任务处于等待却被中断时，该方法抛出 InterruptedException。若 tasks 或其任意元素为 null，抛出 NullPointerException；当没有任何任务可以被调度执行时，会抛出 RejectedExecutionException
<T> T invokeAny(Collection<? extends Callable<T>> tasks, long timeout, TimeUnit unit)	执行给出的 tasks，如果有任务在 timeout（以 unit 时间单元衡量）超时之前成功完成执行（没有抛出异常）就会返回其结果。在超时后还没有完成的任务会被取消。该方法在等待时被中断，则会抛出 InterruptedException。若 tasks 和其任意元素或者 unit 为 null，则抛出 NullPointerException。当 tasks 是空时，抛出 IllegalArgument Exception。若在任意任务完成之前超时，抛出 java.util.concurrent.TimeoutException；当没有任务成功完成，则抛出 ExecutionException；当没有任何任务可以被调度执行时，会抛出 RejectedExecutionException
boolean isShutdown()	当其 executor 已经终止时，返回 true；否则，返回 false
boolean isTerminated()	若紧接着终止之后所有的任务都完成了，那么返回 true；否则，返回 false。该方法绝不会在调用 shutdown()或者 shutdownNow()之前返回 true
void shutdown()	有序地开始关闭之前提交执行的任务，不再接受新的任务。在该 executor 关闭之后调用此方法没有任何作用。该方法不会等待之前提交的任务执行完毕。当需要等待时，使用 awaitTermination()方法
List<Runnable> shutdownNow()	尝试停止所有活跃的执行线程，挂起等待任务的进程，并且返回一组正在等待执行的任务。这里并未保证采用最优的方式停止处理活跃的执行线程。举个例子，通常的实现都会通过 Thread.interrupt()方法取消任务，所以任何没能响应中断的任务可能再也不会终止掉
<T> Future<T> submit Callable<T> task)	提交一个 callable 任务来执行，同时返回一个代表任务等待结果的 Future 实例。这个 Future 实例的 get()方法在成功执行完成后返回任务的结果。当任务无法调度执行，该方法抛出 RejectedExecution Exception。若 task 是空，则抛出 NullPointerException。如果你想在等待任务完成的时候立即阻塞，可以使用这一形式的构造 result=exec.submit(aCallable).get();

续表

方法	说明
Future<?> submit (Runnable task)	提交一个 runnable 任务来执行，同时返回一个代表此任务的 Future 实例。该 Future 的 get 方法在成功完成之后返回 null
<T> Future<T> submit (Runnable task, T result)	提交一个 runnable 任务来执行，同时返回一个 Future 实例，其 get()方法在成功完成的情况下返回结果的值。当任务无法调度执行，该方法抛出 RejectedExecutionException，若 task 为空，则抛出 NullPointerException

表 5-1 引用了枚举类 java.util.concurrent.TimeUnit，它代表给定时间单元上的时间跨度：DAYS、HOURS、MICROSECONDS、MILLISECONDS、MINUTESNANOSECONDS 以及 SECONDS。另外，TimeUnit 还声明了一些方法来做跨单元的转化（比如 long toHours(long duration)），以及在这些单元之上操作定时和延时（比如 void sleep(long timeout)）。

表 5-1 也涉及了 callable 任务。不同于 Runnable 的 void run()方法既不能返回值也不能抛出受检异常，Callable 的 V call()方法不仅可以返回值还可以抛出受检的异常，因为它声明了一个 throws Exception 子句。

最后，表 5-1 涉及 Future 接口，它代表着一种异步计算的结果。这个结果之所以称为 Future，是因为它通常要到未来的某个时刻才有效。Futrue 的泛型是 Futrue<V>，它提供了取消任务、返回任务的结果以及判断任务是否已经结束的方法。表 5-2 描述了 Future 的方法。

表 5-2 Future 接口的方法

方法	说明
boolean cancel(boolean mayInterruptIfRunning)	尝试取消此任务的执行，并且当该任务被取消时，返回 true；否则，返回 false（任务可能在 cancel()方法被调用之前已经正常结束）。当任务已经完成、取消或者由于其他原因无法取消时，取消尝试就会失败。如果取消成功并且该任务还没有启动，这个任务就应该不会再执行了。如果这个任务已经启动，在尝试停止这个任务的过程中，参数 mayInterruptIfRunning 会决定是（true）否（false）应该中断运行

续表

方法	说明
	任务的线程。方法返回之后，后续调用 isDone()方法都会返回 true。当 cancel()方法返回 true 后，isCancelled()方法总会返回 true
V get()	如果需要，则等待此任务完成之后返回结果。当该任务先于此方法被调用被取消，则抛出 java.util.concurrent.CancellationException，若该任务抛出异常则抛出 ExecutionException，并且若当前线程在等待中被中断，则抛出 InterruptedException
V get(long timeout, TimeUnit unit)	最多等待 timeout 时间单元（由 unit 所指定）直到此任务完成执行，之后返回结果（如果可用的话）。当任务在调用此方法之前被取消则抛出 java.util.concurrent.CancellationException，若该任务中途出现异常，则抛出 ExcutionException；若当前线程在等待中被中断，则抛出 InterruptedException；当此方法的 timeout 超时（等待超时时间），则抛出 TimeoutException
boolean isCancelled()	当此任务在正常完成之前被取消了返回 true；否则，返回 false
boolean isDone()	当此任务完成返回 true；否则，返回 false。此方法在正常的中断、出现异常或者取消等场景下都会返回 true

假设你准备写一个应用程序，其图形用户界面可以允许用户输入一个单词。在用户输入单词之后，这个应用程序传递单词到几个在线的辞典并获取每个辞典的条目。这些条目随后会呈现给用户。

因为在线访问会比较慢，且用户界面需要保持响应（或许用户想要结束应用程序），你把“获取单词条目”的任务分摊到一个 executor 上，它在一个单独的线程中运行此任务。下面的例子使用了 ExecutorService、Callable 和 Future 来完成这个目标。

```
ExecutorService executor = ...; // ... represents some executor creation
Future<String[]> taskFuture =
   executor.submit(new Callable<String[]>()
                   {
                   @Override
                   public String[] call()
                   {
```

```
                String[] entries = ...;
                // Access online dictionaries
                // with search word and populate
                // entries with their resulting
                // entries.
                return entries;
            }
        });
// Do stuff.
String entries = taskFuture.get();
```

在以某些方式获得一个 executor 之后（你会很快学会如何做到），例子中的线程将一个 callable 任务提交给了这个 executor。submit()方法会立即返回一个 Future 对象的引用，这个引用控制任务的执行以及结果访问。此线程最终会调用该对象的 get()方法获取结果。

注意：接口 java.util.concurrent.ScheduledExecutorService 继承自接口 ExecutorService。它代表了一种能够让你调度任务运行一次或者在指定延迟之后周期性执行的 exectuor。

尽管你可以实现自己的 Exectuor、ExecutorService 以及 ScheduledExecutorService（例如 class DirectExecutor implements Executor { @Override public void execute(Runnable r) { r.run(); } }——在调用线程中直接运行 executor），不过这还有一个更简单的选项：java.util.concurrent.Executors。

提示：如果你想要创建自己的 ExecutorService 实现，配合使用 java.util.concurrent.AbstractExecutorService 和 java.util.concurrent.FutureTask classes 会很有帮助。

Executors 工具类声明了几个类方法用来返回多种 ExecutorService 和 ScheduledExecutorService 实现的实例（以及其他类型的实例）。这个类的静态方法完成了下列任务：

- 创建并返回一个 ExecutorService 实例，使用常用的配置项设置。
- 创建并返回一个 ScheduledExecutorService 实例，使用常用的配置项设置。
- 创建并返回一个“包装过”的 ExecutorService 或 ScheduledExecutorService 的实例，这个实例让某些方法变得不可访问，从而禁止重新配置 executor service。
- 创建并返回一个 java.util.concurrent.ThreadFactory 的实例（也就是，实现了接口 ThreadFactory 的类实例）来创建新的线程对象。
- 创建并返回一个类闭包的 Callable 实例，它可以用在那些接收 Callable 参数的执行方法中（如 ExecutorService's submit (Callable)方法）。维基百科的“Closure（computer science）”条目 http://en.wikipedia.org/wiki/Closure_(computer_science) 介绍了闭包。

举个例子，static ExecutorService newFixedThreadPool(int nThreads) 方法创建了一个线程池复用固定数量的线程操作一个共享的无限队列。至多有 nThreads 个线程同时处理任务。如果额外的任务在所有线程都活跃时被提交了，它们会在队列中等待一条可用线程。

如果在 executor 停止之前，任何线程因为执行中的失败而终止，而又需要执行后续任务，此时一条新的线程会取代其位置。这些在线程池中的线程会一直存在直到 executor 被显式地关闭。当传递 0 或者负数到 nThreads 的时候，该方法会抛出 IllegalArgumentException。

注意：线程池用来消除不得不为每条提交任务创建一条新线程的开销。线程创建并不廉价，必须创建大量线程可能会严重影响应用程序的性能。

你通常应该会在文件和网络输入、输出上下文中使用 executor、runnable、callable 以及 future，而大量的计算操作提供了另外一种使用这些类型的场景。举个例子，清单 5-1 在欧拉数 e(2.71828...)的计算上下文中使用了一个 executor、callable 以及一个 future。

清单 5-1 计算欧拉数 e

```
import java.math.BigDecimal;
import java.math.MathContext;
import java.math.RoundingMode;

import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

public class CalculateE
{
   final static int LASTITER = 17;

   public static void main(String[] args)
   {
      ExecutorService executor = Executors.newFixedThreadPool(1);
      Callable<BigDecimal> callable;
      callable = new Callable<BigDecimal>()
```

```
        {
            @Override
            public BigDecimal call()
            {
                MathContext mc =
                    new MathContext(100, RoundingMode.HALF_UP);
                BigDecimal result = BigDecimal.ZERO;
                for (int i = 0; i <= LASTITER; i++)
                {
                    BigDecimal factorial =
                        factorial(new BigDecimal(i));
                    BigDecimal res = BigDecimal.ONE.divide(factorial,
                                                           mc);
                    result = result.add(res);
                }
                return result;
            }

            public BigDecimal factorial(BigDecimal n)
            {
                if (n.equals(BigDecimal.ZERO))
                    return BigDecimal.ONE;
                else
                    return n.multiply(factorial(n.
                        subtract(BigDecimal.ONE)));
            }
        };
    Future<BigDecimal> taskFuture = executor.submit(callable);
    try
```

```
    {
        while (!taskFuture.isDone())
            System.out.println("waiting");
        System.out.println(taskFuture.get());
      }
      catch(ExecutionException ee)
      {
        System.err.println("task threw an exception");
        System.err.println(ee);
      }
      catch(InterruptedException ie)
      {
        System.err.println("interrupted while waiting");
      }
      executor.shutdownNow();
    }
}
```

默认主线程首先执行 main()方法，通过调用 Executors 的 newFixed ThreadPool()方法获取一个 executor。之后初始化了一个实现了 Callable 接口的匿名类并且将这个任务提交给 executor，在返回中接收一个 Future 的实例。

在提交任务之后，线程通常会做一些其他工作直到它获取到此项任务的结果。我让主线程在该 Future 实例的 isDone()方法返回 true 之前，不断地打印一条等待中的消息以模拟这种工作（在现实的应用程序中，我不会使用循环）。此时，主线程调用实例的 get()方法来获取结果，稍后才会输出该结果。最后，主线程关闭了这个 executor。

警告： 在 executor 完成之后关闭之是十分重要的。否则，此应用程序可能不会终止。上面的 executor 通过调用 shutdownNow()来完成这一工作（你也可以使用 shutdown()方法）。

这个 callable 的 call()方法通过求数学幂级数 e = 1 / 0! + 1 / 1! + 1 / 2! + ...来计算 e。这一幂级数可以通过 1 / n!求和而获得，n 取值从 0 到无穷（！代表阶乘）。

call()首先初始化 java.math.MathContext 以封装一个精度（小数的位数）和舍位模式。我选择 100 作为 e 精度的上限，同时选择了 HALF_UP 作为舍位模式。

提示： 增加精度和 LASTITER 的值以求更长的收敛时间和更为近似的 e。

call()接下来会把 java.math.BigDecimal 类型的 result 局部变量实例化成 BigDecimal.ZERO。之后进入一个循环计算一个阶乘，用 BigDecimal.ONE 除以这个阶乘，并且把除好的结果加到 result 中去。

divide()方法把 MathContext 实例作为其第 2 个参数来提供舍位信息。（如果我指定 0 作为这个 MathContext 的精度，同时出现一个无限小数部分，例如，除法所得商无法精确地表达——0.3333333...，程序会抛出 java.lang.ArithmeticException，警告调用者这一事实——商无法精确表达。executor 会重新把这个异常作为 ExecutionException 抛出）。

照着下面的样子编译清单 5-1：

```
javac CalculateE.java
```

运行程序：

```
java CalculateE
```

你应该能观测到和下面类似的输出（你很可能观测到更多的 waiting 消息）：

```
waiting
waiting
waiting
waiting
waiting
2.71828182845904507051604779584860506117897963525103269890073500406522504250
484331405588797434424574173003945406271l
```

5.3 练习

下面的练习用来验证你对第 5 章内容的掌握程度。

1. 并发工具类是什么？
2. 指出并发工具类型位于哪些包下面。
3. 给出任务的定义。
4. 给出 executor 的定义。
5. 说明 Executor 接口的局限。
6. 如何克服 Executor 的局限。
7. Runnable 的 run()方法和 Callable 的 call()方法存在什么区别？
8. 判断对错：你可以从 Runnable 的 run()方法中抛出受检和非受检的异常，但是只能从 Callable 的 call()方法中抛出非受检的异常。
9. 给出 future 的定义。

10. 描述类 Executors 的 newFixedThreadPool()方法。

11. 使用 Executors 和 ExecutorService 来重构下面的 CountingThreads 应用程序。

```
public class CountingThreads
{
   public static void main(String[] args)
   {
      Runnable r = new Runnable()
                {
                  @Override
                  public void run()
                  {
                    String name = Thread.currentThread().
                                        getName();
                    int count = 0;
                    while (true)
                      System.out.println(name + ": " +
                                           count++);
                  }
                };
      Thread thdA = new Thread(r);
      Thread thdB = new Thread(r);
      thdA.start();
      thdB.start();
   }
}
```

12. 当你执行前面练习中 CountingThreads 应用程序时，你会观测到诸如 pool-1-thread-1 所标识的线程输出。请修改 CountingThreads，以便可以观测名称 A 和 B。提示：你需要使用 ThreadFactory。

5.4 小结

Java 对底层线程操作的支持使得你可以创建多线程应用程序。这些程序比对应的单线程程序提供了更好的性能和响应能力。不过，影响应用程序可扩展性的性能及其他问题，促使 Java 5 引入了并发工具类。

这个并发工具类将多种类型组织进了 3 个包中：java.util.concurrent、java.util.concurrent.atomic 以及 java.util.concurrent.locks。基础类型如 executor、线程池、并发 hashmap 以及其他高级的并发构造被存储在 java.util.concurrent 中；支持在单个变量上进行无锁、线程安全编程的类存储在 java.util.concurrent.atomic 中；在某些条件上获取锁或者执行等待的类型存储在 java.util.concurrent.locks 中。

Executor 框架从任务执行中解耦了任务的提交操作，它包含 Executor、ExecutorService 以及 ScheduledExecutorService。你可以通过调用类 Executors 中任意一个工具方法来获取一个 executor。另外，Executors 和 callable 及 future 相关联。

第 6 章会涉及同步器。

第6章

同步器

Java 提供了 synchronized 关键字对临界区进行线程同步访问。由于基于 synchronized 很难正确地编写同步代码，并发工具类提供了高级的同步器（控制通用同步方法的类）。在本章中，我会介绍倒计时门闩（countdown latch）、同步屏障（cyclic barrier）、交换器（exchanger）、信号量（semaphore）以及 phaser 同步器。

6.1 倒计时门闩

倒计时门闩会导致一条或多条线程在“门口”一直等待，直到另一条线程打开这扇门，线程才得以继续运行。它是由一个计数变量和两个操作组成的，这两个操作分别是“导致一条线程等待直到计数变为 0”以及“递减计数变量”。

类 java.util.concurrent.CountDownLatch 实现了倒计时门闩同步器。你可以通过调用这个类的构造方法 CountDownLatch (int count)并指定计数个数，来初始化一个 CountDownLatch 的实例。当 count 的值是负数，该方法会抛出 java.lang.IllegalArgumentException。

类 CountDownLatch 也提供了下列方法：

- void await()：除非线程被中断，否则强制调用线程一直等到计数倒数到 0。线程被中断时会抛出 java.lang.InterruptedException。当 count 是 0 时，该方法立即返回。
- boolean await(long timeout, TimeUnit unit)：除非线程被中断，否则强制调用线程一直等到计数倒数至 0 或者以 unit 作为时间单元的 timeout 超时。线程被中断时会抛出 java.lang.InterruptedException。当 count 是 0 时，该方法立即返回 true；当超过等待时间时，返回 false。
- void countDown()：递减计数，当计数降至 0 时，释放所有等待线程。当该方法被调用时 count 已经为 0，那么什么也不会发生。
- long getCount()：返回当前的计数，该方法对于测试和调试很有用。
- String toString()：返回一条标识这个门闩及其状态的字符串。状态是用中括号括住的字符串常量"Count = "拼接上当前的计数。

你会经常使用一个倒计时门闩来保证多条线程几乎同时开始工作。举个例子，请看清单 6-1。

清单 6-1 倒计时门闩示例

```
import java.util.concurrent.CountDownLatch;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class CountDownLatchDemo
{
   final static int NTHREADS = 3;
```

```
public static void main(String[] args)
{
   final CountDownLatch startSignal = new CountDownLatch(1);
   final CountDownLatch doneSignal = new CountDownLatch(NTHREADS);
   Runnable r = new Runnable()
                {
                   @Override
                   public void run()
                   {
                      try
                      {
                         report("entered run()");
                         startSignal.await();  // wait until told to ...
                         report("doing work"); // ... proceed
                         Thread.sleep((int) (Math.random() * 1000));
                         doneSignal.countDown(); // reduce count on which
                                                 // main thread is ...
                                                 // waiting
                      }
                      catch (InterruptedException ie)
                      {
                         System.err.println(ie);
                      }
                   }
                   void report(String s)
                   {
                      System.out.println(System.currentTimeMillis() +
                                         ": " + Thread.currentThread() +
```

```
                                    ": " + s);
                }
            };
        ExecutorService executor = Executors.newFixedThreadPool(NTHREADS);
        for (int i = 0; i < NTHREADS; i++)
            executor.execute(r);
        try
        {
            System.out.println("main thread doing something");
            Thread.sleep(1000); // sleep for 1 second
            startSignal.countDown(); // let all threads proceed
            System.out.println("main thread doing something else");
            doneSignal.await(); // wait for all threads to finish
            executor.shutdownNow();
        }
        catch (InterruptedException ie)
        {
            System.err.println(ie);
        }
    }
}
```

清单 6-1 中的默认主线程首先创建了一对倒计时门闩。这个 **startSignal** 门闩会在默认主线程就绪之前禁止任何工作线程执行，而 **doneSignal** 门闩会使得默认主线程等待所有的工作线程全部结束执行。

默认主线程接下来创建了一个带有 **run()**方法的 **runnable**，它会被后续创建的工作线程执行。

run()方法首先输出一条消息，然后调用 **startSignal** 的 **await()**方法，

在继续执行之前会一直等待直到门闩的计数变为 0。继续执行时，run()方法输出一条表示工作正在进行的消息，并且挑一个随机的时间（从 0 到 999 毫秒）睡眠用于模拟正在做的工作。

到这里，run()调用了 doneSignal 的 countDown()方法递减了门闩的计数。一旦计数降到 0，等待这个信号的默认主线程就会继续执行，关闭 executor 和中断应用程序。

创建好 runnable 之后，从包含 NTHREADS 条线程的线程池获取一个 executor，然后在这个 executor 上连续调用 NTHREADS 次 execute()方法，依次把 runnable 传入线程池中。这一动作会启动线程运行 run()方法。

接下来，默认主线程输出一条消息并且通过睡眠 1 秒的方式模拟执行其他的工作（得以让所有工作线程依次执行 run()方法，进而调用 startSignal.await()），然后调用 startSingal 的 countDown()方法使工作线程开始运行，再输出一条消息表示主线程正在做一些别的事情，接着调用 doneSignal 的 await()方法，在它可以继续执行之前等待倒计时门闩的计数降至 0。

照着下面编译清单 6-1：

```
javac CountDownLatchDemo.java
```

运行程序：

```
java CountDownLatchDemo
```

你应该能观测到类似下面的输出（消息顺序可能略有出入）：

```
main thread doing something
1445802274931: Thread[pool-1-thread-2,5,main]: entered run()
1445802274931: Thread[pool-1-thread-3,5,main]: entered run()
1445802274931: Thread[pool-1-thread-1,5,main]: entered run()
main thread doing something else
```

```
1445802275931: Thread[pool-1-thread-2,5,main]: doing work
1445802275931: Thread[pool-1-thread-3,5,main]: doing work
1445802275933: Thread[pool-1-thread-1,5,main]: doing work
```

6.2 同步屏障

同步屏障允许一组线程彼此互相等待，直到抵达某个公共的屏障点。因为该屏障在等待线程被释放之后可以重用，所以称它为可循环使用的屏障。该同步器对于数量固定并且互相之间必须不时等待彼此的多线程应用很有用。

类 java.util.concurrent.CyclicBarrier 实现了同步屏障。通过调用这个类的 CyclicBarrier(int parties)构造函数，你可以初始化一个包含指定 parties（拥有共同执行目标的多条线程）数目的 CyclicBarrier 实例。如果 parties 的值小于 1，构造函数就会抛出 IllegalArgumentException。

另外，你可以调用 CyclicBarrier(int parties,Runnable barrierAction)构造函数来初始化一个同步屏障，包含指定 parties 数目的线程以及一旦跨越屏障就会执行的 barrierAction。换句话说，当 parties-1 条线程处于等待之中并且又有一条线程到达，这条到达的线程会执行 barrierAction，然后所有的线程继续执行。这个 runnable 适用于在任意线程继续执行之前更新共享状态。当传递给 parties 的值小于 1，该构造函数抛出 IllegalArgumentException（前面的构造函数会调用这个构造函数，若把 barrierAction 设为 null，那么当跨越屏障时，就没有 runnable 可供执行）。

CyclicBarrier 也提供了下列方法：

- int await()：强制调用线程一直等待直到所有的 parties 都已经在同步屏障上调用了 await()方法。当调用线程自己或其他等待线程被中断、有线程在等待中超时或者有线程在同步屏障之上调用 reset()方法，该调用线程就会停止等待。如果调用线程在进入方法时设置过中断的状态或者等待时被中断，该方法就会抛出 InterruptedException 并且其中断状态会被清除。当有线程正在等待时该同步屏障被重置了（通过 reset()方法），或者当同步屏障在 await()方法被调用或任意线程正在等待时同步屏障被打破，该方法就会抛出 java.util.concurrent.BrokenBarrierException。一旦有线程在等待时被中断，其他所有等待中的线程都会抛出 BrokenBarrierException 并且同步屏障也会被设置为打破状态。如果调用线程是最后一条到达的线程，并且构造函数中提供了一个非空的 barrierAction，这条线程就会在允许其他线程继续执行之前率先执行这个 runnable。该方法会返回调用线程的到达索引，getParties() - 1 代表第一条到达的线程，0 代表最后一条到达的线程。
- int await(long timeout, TimeUnit unit)：除了让你指定调用线程愿意等待的时长之外，该方法等同于之前的方法。当线程在等待中超时，该方法会抛出 java.util.concurrent.TimeoutException。
- int getNumberWaiting()：返回当前在同步屏障上等待的线程数目。该方法对于调试以及断言十分有用。
- int getParties()：返回需要跨越同步屏障的线程数目。
- boolean isBroken()：当一条或者多条线程由于在同步屏障创建或上次重置之后，中断或超时从而打破同步屏障，又或者因为一个异常导致 barrier action 失败时，返回 true；否则返回 false。

- `void reset()`：把同步屏障重置到其原始状态。如果此时任意的线程等待在这个同步屏障上，就会抛出一个 `BrokenBarrierException`。注意，在由于某些原因发生的跳出操作之后进行重置是非常难以实现的。线程需要通过一些其他的方式重新同步并挑选一条线程来进行重置操作。所以，最好还是给后续的使用创建一个新的同步屏障。

同步屏障在并行分解的场合下很有用。在这里长时间的任务被分割成多个子任务，它们单独的结果随后会被合并到整个任务的结果当中。同步屏障的 Javadoc 给出了清单 6-2 的示例代码。

清单 6-2　使用同步屏障把一个任务分解成多个子任务

```
import java.util.concurrent.BrokenBarrierException;
import java.util.concurrent.CyclicBarrier;

public class CyclicBarrierDemo
{
   public static void main(String[] args)
   {
      float[][] matrix = new float[3][3];
      int counter = 0;
      for (int row = 0; row < matrix.length; row++)
         for (int col = 0; col < matrix[0].length; col++)
         matrix[row][col] = counter++;
      dump(matrix);
      System.out.println();
      Solver solver = new Solver(matrix);
      System.out.println();
      dump(matrix);
```

```
   }

   static void dump(float[][] matrix)
   {
      for (int row = 0; row < matrix.length; row++)
      {
         for (int col = 0; col < matrix[0].length; col++)
            System.out.print(matrix[row][col] + " ");
         System.out.println();
      }
   }
}

class Solver
{
   final int N;
   final float[][] data;
   final CyclicBarrier barrier;

   class Worker implements Runnable
   {
      int myRow;
      boolean done = false;

      Worker(int row)
      {
         myRow = row;
      }

      boolean done()
```

```
{
    return done;
}

void processRow(int myRow)
{
    System.out.println("Processing row: " + myRow);
    for (int i = 0; i < N; i++)
        data[myRow][i] *= 10;
    done = true;
}

@Override
public void run()
{
    while (!done())
    {
        processRow(myRow);

        try
        {
            barrier.await();
        }
        catch (InterruptedException ie)
        {
            return;
        }
        catch (BrokenBarrierException bbe)
        {
```

```
                return;
            }
        }
    }
}

public Solver(float[][] matrix)
{
    data = matrix;
    N = matrix.length;
    barrier = new CyclicBarrier(N,
                                new Runnable()
                                {
                                    @Override
                                    public void run()
                                    {
                                        mergeRows();
                                    }
                                });

    for (int i = 0; i < N; ++i)
        new Thread(new Worker(i)).start();

    waitUntilDone();
}

void mergeRows()
{
    System.out.println("merging");
    synchronized("abc")
```

```
        {
            "abc".notify();
        }
    }

    void waitUntilDone()
    {
        synchronized("abc")
        {
            try {
                System.out.println("main thread waiting");
                "abc".wait();
                System.out.println("main thread notified");
            }
            catch (InterruptedException ie)
            {
                System.out.println("main thread interrupted");
            }
        }
    }
}
```

清单 6-2 中的默认主线程首先创建了一个浮点数矩阵并把这个矩阵导出到标准输出流当中。主线程然后初始化了类 `Solver`，这个类为每一行分别创建一条线程进行计算。更改之后的矩阵随后也被导出。

`Solver` 提供了一个构造函数，它接收矩阵参数并用属性 `data` 指向这个矩阵，同时用属性 `N` 指向矩阵行的数目。这个构造函数之后创建了一个拥有 `N` 条线程的同步屏障，并且又负责把所有的行合并到矩阵。最后，该构造函数分别创建工作线程，负责处理矩阵中单一的行。之后构造函数等待所有工作线程结束。

工作线程的 run()方法反复在指定的行上调用 processRow()方法直到 done()方法返回 true。(在这个例子当中)在 processRow()执行过一次之后，done()就会返回 true。在 processRow()返回之后，也就意味着矩阵行已经被处理了，当前的工作线程就会于同步屏障之上调用所有 await()方法，它也就无法执行下去了。

某些时候，所有的工作线程都已经调用 await()方法了。当处理矩阵最后一行的这条最后的线程调用了 await()方法，它就会触发屏障动作来将所有处理过的矩阵行合并到最终的矩阵当中。在这个例子中，合并操作不是必须的，但在更复杂的例子里或许是必要的。

最后 mergeRow()方法进行的任务是去通知调用 Solver 构造函数的主线程。主线程在 String 对象"abc"所关联的监听器上等待，而调用 notify()足以唤醒在此监听器上等待这唯一一条的线程。

照下面编译清单 6-2：

```
javac CyclicBarrierDemo.java
```

运行程序：

```
java CyclicBarrierDemo
```

你应该能观测到类似下面的输出（消息的顺序可能略有出入）：

```
0.0 1.0 2.0
3.0 4.0 5.0
6.0 7.0 8.0

main thread waiting
Processing row: 0
Processing row: 1
Processing row: 2
```

```
merging
main thread notified

0.0 10.0 20.0
30.0 40.0 50.0
60.0 70.0 80.0
```

6.3 交换器

交换器提供了一个线程彼此之间能够交换对象的同步点。每条线程都会往这个交换器的 exchange()方法传入一些对象，匹配伙伴线程，同时接收伙伴线程中的对象作为返回值。在诸如遗传算法（http://en.wikipedia.org/wiki/Genetic_algorithm）和管道设计的应用程序中，交换器会很有用。

泛型类 java.util.concurrent.Exchanger<V>实现了交换器。你可以通过调用 Exchanger()构造函数来初始化一个交换器，然后调用下列方法之一进行一次交换。

- V exchange(V x)：在这个交互点上等待其他线程到达（除非调用线程被中断了），之后将所给对象传入其中，接收其他线程的对象作为返回。如果其他的线程已经在交换点上等待，为了线程调度它会从中恢复并且会接收调用线程所传入的对象。当前线程会立即返回，接收其他线程传入交换器中的对象。当调用线程被中断了，该方法会抛出 InterruptedException。
- V exchange(V x, long timeout, TimeUnit unit)：除了让你指定调用线程愿意等待的时长之外，该方法等同于之前的方法。当线程

在等待中超时，该方法会抛出 TimeoutException。

清单6-3扩展了这个Exchanger的Javadoc中重复地填充和清空Exchanger缓冲区的例子。

清单 6-3 使用一个交换器来交换缓冲区

```
import java.util.ArrayList;
import java.util.List;

import java.util.concurrent.Exchanger;

public class ExchangerDemo
{
   final static Exchanger<DataBuffer> exchanger =
      new Exchanger<DataBuffer>();

   final static DataBuffer initialEmptyBuffer = new DataBuffer();
   final static DataBuffer initialFullBuffer = new DataBuffer("I");

   public static void main(String[] args)
   {
      class FillingLoop implements Runnable
      {
         int count = 0;

         @Override
         public void run()
         {
            DataBuffer currentBuffer = initialEmptyBuffer;
```

```
        try
        {
            while (true)
            {
                addToBuffer(currentBuffer);
                if (currentBuffer.isFull())
                {
                    System.out.println("filling thread wants to exchange");
                    currentBuffer = exchanger.exchange(currentBuffer);
                    System.out.println("filling thread receives exchange");
                }
            }
        }
        catch (InterruptedException ie)
        {
            System.out.println("filling thread interrupted");
        }
    }

    void addToBuffer(DataBuffer buffer)
    {
        String item = "NI" + count++;
        System.out.println("Adding: " + item);
        buffer.add(item);
    }
}

class EmptyingLoop implements Runnable
```

```
{
    @Override
    public void run()
    {
        DataBuffer currentBuffer = initialFullBuffer;
        try
        {
            while (true)
            {
                takeFromBuffer(currentBuffer);
                if (currentBuffer.isEmpty())
                {
                    System.out.println("emptying thread wants to " +
                                       "exchange");
                    currentBuffer = exchanger.exchange(currentBuffer);
                    System.out.println("emptying thread receives " +
                                       "exchange");
                }
            }
        }
        catch (InterruptedException ie)
        {
            System.out.println("emptying thread interrupted");
        }
    }
    void takeFromBuffer(DataBuffer buffer)
    {
        System.out.println("taking: " + buffer.remove());
```

```
        }
      }
      new Thread(new EmptyingLoop()).start();
      new Thread(new FillingLoop()).start();
    }
}

class DataBuffer
{
  private final static int MAXITEMS = 10;

  private final List<String> items = new ArrayList<>();

  DataBuffer()
  {
  }

  DataBuffer(String prefix)
  {
    for (int i = 0; i < MAXITEMS; i++)
    {
      String item = prefix + i;
      System.out.printf("Adding %s%n", item);
      items.add(item);
    }
  }

  synchronized void add(String s)
```

```
    {
        if (!isFull())
            items.add(s);
    }

    synchronized boolean isEmpty()
    {
        return items.size() == 0;
    }

    synchronized boolean isFull()
    {
        return items.size() == MAXITEMS;
    }

    synchronized String remove()
    {
        if (!isEmpty())
            return items.remove(0);
        return null;
    }
}
```

清单 6-3 的默认主线程通过静态属性初始化创建了一个交换器以及一对缓冲区。之后，它初始化本地的类 EmptyingLoop 和 FillingLoop，并将这些 runnable 传递到新的线程实例当中，这些实例随后会被启动。（也可以使用 executors）每个 runnables 的 run()方法进入一个无限循环，反复地往它的

缓冲中添加或者删除。当缓冲区满了或者空了，这个交换器会用来交换这些缓冲，持续地添加或清空。

照下面编译清单 6-3：

```
javac ExchangerDemo.java
```

运行程序：

```
java ExchangerDemo
```

你应该能观测到类似下面的输出，这里只列出了开始部分（消息的顺序可能略有出入）。

```
Adding I0
Adding I1
Adding I2
Adding I3
Adding I4
Adding I5
Adding I6
Adding I7
Adding I8
Adding I9
taking: I0
taking: I1
taking: I2
taking: I3
taking: I4
taking: I5
taking: I6
taking: I7
taking: I8
```

```
taking: I9
emptying thread wants to exchange
Adding: NI0
Adding: NI1
Adding: NI2
Adding: NI3
Adding: NI4
Adding: NI5
Adding: NI6
Adding: NI7
Adding: NI8
Adding: NI9
filling thread wants to exchange
filling thread receives exchange
emptying thread receives exchange
Adding: NI10
taking: NI0
Adding: NI11
taking: NI1
Adding: NI12
```

6.4 信号量

信号量维护了一组许可证（permit），以约束访问被限制资源的线程数。当没有可用的许可证时，线程的获取尝试会一直阻塞，直到其他的线程释放一个许可证。

■ **注意**：当前的值可以被递增 1 的信号量称作计数信号量。而当前的值只能取 0 或 1 的信号量则称为二进制信号量或者互斥信号量。在这两种场景中，当前的值都不能为负。

类 java.util.concurrent.Semaphore 实现了这一同步器，同时将信号量概念化成一个维护一组许可证的对象。你可以调用 Semaphore(int permits) 构造函数来初始化一个信号量，其中 permits 指定了可用许可证的数量。这个信号量的公平策略被设置成 false（不公平）。或者，你也可以调用 Semaphore(int permits, boolean fair)构造函数把信号量的公平策略设置成 true（公平）。

6.5 信号量和公平策略

当公平策略设置为 false，信号量不会保证线程获取许可证的顺序。特别地，抢占是允许的。也就是说，即便线程已经在等待，调用了 acquire()方法的新线程还是能先于这条线程被分配许可证。逻辑上，新线程把自己放到了等待线程队列的队首了。当公平策略被设置为 true，信号量就能保证调用 acquire()方法的任意线程能按照方法被调用处理的顺序获取许可证（先进先出，FIFO）。因为 FIFO 顺序需要应用到这些方法中指定的内部执行点上，很可能一条线程先于另一条线程调用 acquire()方法，但是却后于那条线程抵达顺序点，从方法中返回也类似。当然，不限时的 tryAcquire()方法不会遵循公平策略的设定——它们会获取任意可用的许可证。

一般来讲，信号量通常用来控制资源访问，它应当初始化成公平的，从而保证不会有任何线程在访问资源时饿死。当针对其他同步控制使用信号量的时候，不公平策略带来的吞吐量好处是超过公平策略的。

信号量也提供下列方法：

- `void acquire()`：从这个信号量中获取一个许可证，否则阻塞直到有一个许可证可用或者调用线程被中断。当调用线程中断，抛出`InterruptedException`。
- `void acquire(int permits)`：从这个信号量中获取`permits`数量的许可证，否则阻塞直到这些许可证可用或者调用线程中断了。当调用线程中断，则抛出`InterruptedException`；当`permits`小于0的时候，抛出`IllegalArgumentException`。
- `void acquireUninterruptibly()`：获取一个许可证，否则阻塞直到有一个许可证可用。
- `void acquireUninterruptibly(int permits)`：获取`permits`个许可证，否则阻塞直到这些许可证可用。当`permits`小于0，则抛出`IllegalArgumentException`。
- `int availablePermits()`：返回当前可用许可证的数目。该方法适用于调试和测试。
- `int drainPermits()`：获取并返回立即可用的许可证的数量。
- `int getQueueLength()`：返回等待获取许可证的大致线程数。由于线程数在该方法遍历内部数据结构的时候可能会动态改变，返回的值只能是估算值。该方法设计用来监控系统状态而不是用来做同步控制。
- `boolean hasQueuedThreads()`：查询是否存在等待获取许可证的线程。由于取消可能发生在任意时刻，返回`true`并不能保证其他线程

就能获取许可证。该方法主要设计用于监控系统状态。当可能存在其他等待线程的时候，它会返回 true。

- boolean isFair()：返回公平性设置（公平返回 true，不公平返回 false）。
- void release()：释放一个许可证，将其返回给信号量。可用许可证的数目增加 1。如果任何线程正在尝试获取一个许可证，被选到的线程就会被给予刚刚释放的许可证。那条线程就会因为线程调度而被重新启用。
- void release(int permits)：释放 permits 数量的许可证，将它们返回给信号量。可用许可证的数目增加 permits 个。如果任意线程正在尝试获取许可证，被选到的线程就会被给予刚刚释放的许可证。如果可用许可证的数目满足了线程的要求，该线程就会由于线程调度而被重新启用。否则，该线程就会一直等待，直到足够的许可证可用。如果在满足该线程的要求之后仍有许可证可用，那些许可证就会被分派到其他尝试获取许可证的线程。当 permits 小于 0 的时候，会抛出 IllegalArgumentException。
- String toString()：返回标识该信号量及其状态的字符串。该状态用括号括住，包含了字符串常量"Permits = "，后面跟着许可证的数目。
- boolean tryAcquire()：仅当调用时有一个许可证可用的情况，才能从这个信号量中获取这个许可证。当获取许可证后返回 true；否则，立刻返回 false。
- boolean tryAcquire(int permits)：仅当调用时有 permits 个许可证可用的情况，才能从这个信号量中获取这些许可证。当获取到这些许可证后，返回 true，否则立即返回 false。当 permits 小于 0 时，抛出 IllegalArgumentException。

- boolean tryAcquire(int permits, long timeout, TimeUnit unit)：和前面的方法类似，不过当 permits 个许可证尚不可用时，调用线程会等待。当全部许可证可用时等待结束。若 timeout 超时或者调用线程中断，则抛出 InterruptedException。
- boolean tryAcquire(long timeout TimeUnit unit)：和 tryAcquire(int permits)方法类似，不过调用线程会一直等待直到有一个许可证可用。当许可证可用时，等待结束。若 timeout 超时或者调用线程中断，则抛出 InterruptedException。

清单 6-4 基于 Semaphore Javadoc 给出的“控制对于一组条目的访问”例子进行了扩展。

清单 6-4 使用计数信号量去控制对一组条目的访问

```
import java.util.concurrent.Executors;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Semaphore;

public class SemaphoreDemo
{
   public static void main(String[] args)
   {
      final Pool pool = new Pool();
      Runnable r = new Runnable()
                   {
                      @Override
                      public void run()
                      {
                         String name = Thread.currentThread().getName();
```

```
                try
                {
                   while (true)
                   {
                      String item;
                      System.out.println(name + " acquiring " +
                                                (item = pool.getItem()));
                      Thread.sleep(200 +
                                   (int) (Math.random() * 100));
                      System.out.println(name + " putting back " +
                                                item);
                      pool.putItem(item);
                   }
                }
                catch (InterruptedException ie)
                {
                   System.out.println(name + "interrupted");
                }
             }
          };
      ExecutorService[] executors =
         new ExecutorService[Pool.MAX_AVAILABLE + 1];
      for (int i = 0; i < executors.length; i++)
      {
         executors[i] = Executors.newSingleThreadExecutor();
         executors[i].execute(r);
      }
   }
```

```
}

final class Pool
{
   public static final int MAX_AVAILABLE = 10;

   private final Semaphore available = new Semaphore(MAX_AVAILABLE, true);

   private final String[] items;

   private final boolean[] used = new boolean[MAX_AVAILABLE];

   Pool()
   {
      items = new String[MAX_AVAILABLE];
      for (int i = 0; i < items.length; i++)
         items[i] = "I" + i;
   }

   String getItem() throws InterruptedException
   {
      available.acquire();
      return getNextAvailableItem();
   }

   void putItem(String item)
   {
      if (markAsUnused(item))
```

```
        available.release();
    }

    private synchronized String getNextAvailableItem()
    {
        for (int i = 0; i < MAX_AVAILABLE; ++i)
        {
            if (!used[i])
            {
                used[i] = true;
                return items[i];
            }
        }
        return null; // not reached
    }

    private synchronized boolean markAsUnused(String item)
    {
        for (int i = 0; i < MAX_AVAILABLE; ++i)
        {
            if (item == items[i])
            {
                if (used[i])
                {
                    used[i] = false;
                    return true;
                }
                else
                    return false;
```

```
            }
        }
        return false;
    }
}
```

清单 6-4 中，默认主线程创建了一个资源池、一个反复获取和归还资源的 runnable 和一组 executors。每个执行者都会执行这一 runnable。

类 Pool 的 String getItem()以及 void putItem(String item)方法获取和归还基于字符串的资源。在通过 getItem()获取一个条目之前，调用线程必须从信号量中获取一个许可证，这样才能保证这个条目可用。当线程处理完这个条目，它会去调用 putItem(String)方法，该方法会将此条目归还到池中，之后会释放一个针对这个信号量的许可证。这样，便可让其他的线程获取到那个条目。

当 acquire()方法被调用的时候并不会持有同步锁，因为那样会阻止条目被归还到池中。不过，String getNextAvailableItem()和 boolean markAsUnused(String item)方法会同步地去维持池的一致性（这个信号量把对资源池限制访问和对需要维护资源池一致性的同步操作都独立封装起来了）。

照下面编译清单 6-4：

```
javac SemaphoreDemo.java
```

运行程序：

```
java SemaphoreDemo
```

你应该能观测到类似下面的输出，这里只给出了输出的开始部分（消息的顺序可能略有出入）：

```
pool-1-thread-1 acquiring I0
pool-2-thread-1 acquiring I1
pool-3-thread-1 acquiring I2
pool-5-thread-1 acquiring I3
pool-7-thread-1 acquiring I4
pool-4-thread-1 acquiring I5
pool-6-thread-1 acquiring I6
pool-9-thread-1 acquiring I7
pool-8-thread-1 acquiring I8
pool-10-thread-1 acquiring I9
pool-9-thread-1 putting back I7
pool-2-thread-1 putting back I1
pool-11-thread-1 acquiring I7
pool-9-thread-1 acquiring I1
pool-8-thread-1 putting back I8
pool-2-thread-1 acquiring I8
pool-5-thread-1 putting back I3
pool-8-thread-1 acquiring I3
pool-4-thread-1 putting back I5
pool-5-thread-1 acquiring I5
pool-6-thread-1 putting back I6
pool-4-thread-1 acquiring I6
pool-1-thread-1 putting back I0
pool-6-thread-1 acquiring I0
pool-7-thread-1 putting back I4
pool-1-thread-1 acquiring I4
pool-10-thread-1 putting back I9
pool-7-thread-1 acquiring I9
```

```
pool-3-thread-1 putting back I2
pool-10-thread-1 acquiring I2
```

6.6 Phaser

phaser 是一个更加弹性的同步屏障。和同步屏障一样，一个 phaser 使得一组线程在屏障上等待，在最后一条线程到达之后，这些线程得以继续执行。phaser 也提供 barrier action 的等价操作。和同步屏障协调固定数目的线程不同，一个 phaser 能够协调不定数目的线程，这些线程可以在任何时候注册。为了实现这一功能，phaser 使用了 phase 和 phase 值。

phase 是 phaser 当前的状态，同时这一状态被一个整型的 phase 值所确定。当最后一条注册的线程到达 phaser 屏障，phaser 提前抵达 phase 并且给其加 1。

类 java.util.concurrent 实现了 phaser。由于 Javadoc 中已经详细描述了这个类，我这里只会介绍一些构造函数和方法：

- Phaser(int parties)构造函数创建了一个 phaser，一开始就协调 parties 数目的线程（还没有抵达 phaser 屏障），同时其 phase 初始为 0。
- int register()方法往这个 phaser 中添加一条尚未抵达的线程，同时返回 phase 值作抵达分类用。这个值称为抵达 phase 值。
- int arriveAndAwaitAdvance()方法记录到达并等待 phaser 前进（在其他线程已经到达之后开始）。它会返回抵达 phase 值。
- int arriveAndDeregister()方法抵达此 phaser，同时从中注销而不会等待其他线程到达，由此减少了未来 phase 上需要前进的线程数量。

清单 6-5 基于 Phaser Javadoc 的第一个例子，提供了一个关于 phaser 同步器的示例。

清单 6-5 使用一个 Phaser 来控制一个一次性的动作，该动作作用于可变数量的线程上。

```
import java.util.ArrayList;
import java.util.List;

import java.util.concurrent.Executors;
import java.util.concurrent.Phaser;

public class PhaserDemo
{
   public static void main(String[] args)
   {
      List<Runnable> tasks = new ArrayList<>();
      tasks.add(() -> System.out.printf("%s running at %d%n",
                                        Thread.currentThread().getName(),
                                        System.currentTimeMillis()));

      tasks.add(() -> System.out.printf("%s running at %d%n",
                                        Thread.currentThread().getName(),
                                        System.currentTimeMillis()));
      runTasks(tasks);
   }

   static void runTasks(List<Runnable> tasks)
   {
      final Phaser phaser = new Phaser(1); // "1" (register self)
```

```
        // create and start threads
        for (final Runnable task: tasks)
        {
            phaser.register();
            Runnable r = () ->
                         {
                            try
                            {
                               Thread.sleep(50 + (int) (Math.random() * 300));
                            }
                            catch (InterruptedException ie)
                            {
                               System.out.println("interrupted thread");
                            }
                            phaser.arriveAndAwaitAdvance(); // await the ...
                                                            // creation of ...
                                                            // all tasks
                            task.run();
                         };
            Executors.newSingleThreadExecutor().execute(r);
        }
        // allow threads to start and deregister self
        phaser.arriveAndDeregister();
    }
}
```

清单 6-5 中默认主线程创建了一对 runnable 任务，每个任务报告它们自己开始运行的时间（以毫秒记）。在创建一个 Phaser 的实例之后，运行这些任务并等待全部任务到达该屏障。

照下面这样编译清单 6-5：

```
javac PhaserDemo.java
```

运行程序：

```
java PhaserDemo
```

你应该能观测到类似下列的输出（这个应用程序不会终止——按下 Ctrl+C 或者相应的触发键来终止程序）：

```
pool-1-thread-1 running at 1445806012709
pool-2-thread-1 running at 1445806012712
```

就和你期待的倒计时门闩的行为一样，尽管由于 `Thread.sleep()`方法的存在，一条线程可能延迟 349 毫秒之多，但是全部线程（在这个例子当中）都在同样的时间开始运行。

注释掉 `phaser.arriveAndAwaitAdvance(); // await the ...`，你现在应该能观测到这些线程完全不同时运行了，显示如下：

```
pool-2-thread-1 running at 1445806212870
pool-1-thread-1 running at 1445806213013
```

6.7 练习

下面的练习用来验证你对第 6 章内容的掌握程度。

1. 给出同步器的定义。

2. 描述倒计时门闩（CountDownLatch）的行为。

3. 当调用 `CountDownLatch` 的 `void countDown()`方法且计数变为 0 时会发生什么呢？

4. 描述同步屏障的行为。

5. 判断对错：任意线程等待时屏障被重置或者在调用 await()方法时屏障被打破，则 CyclicBarrier 的 int await()方法返回-1。

6. 描述一个交换器的行为。

7. Exchanger 的 V exchange(V x)方法完成了什么操作？

8. 描述一个信号量的行为。

9. 指出两种类型的信号量。

10. 描述一个 phaser 的行为。

11. Phaser 的 int register()方法返回什么？

12. 清单 3-2（第 3 章中）展示了一个改良版的 PC 程序。重新使用类 Semaphore 处理同步来创建这一应用程序。

6.8 小结

Java 提供了 synchronized 关键字对临界区进行线程同步访问。由于基于 synchronized 很难正确地编写同步代码，并发工具类提供了高级的同步器(控制通用同步方法的类)。

倒计时门闩会导致一条或多条线程在“门口”一直等待，直到另一条线程打开这扇门，线程才得以继续运行。它是由一个计数变量和两个操作组成的，这两个操作分别是“导致一条线程等待直到计数变为 0”以及“递减计数变量”。

同步屏障使得一组线程一直等待彼此抵达一个公共的屏障点。由于在等待线程被释放之后可以被复用，它是可回收使用的。这个同步器在涉及固定规模的且必须不时等待彼此的线程中很有用。

同步屏障允许一组线程彼此互相等待，直到抵达到某个公共的屏障点。因为该屏障在等待线程被释放之后可以重用，所以称它为可循环使用的屏障。该同步器对于数量固定并且互相之间必须不时等待彼此的多线程应用很有用。

信号量维护了一组许可证（permit），以约束访问被限制资源的线程数。当没有可用的许可证时，线程的获取尝试会一直阻塞，直到其他的线程释放一个许可证。

phaser 是一个更加弹性的同步屏障。和同步屏障一样，一个 phaser 使得一组线程在屏障上等待，在最后一条线程到达之后，这些线程得以继续执行。phaser 也提供 barrier action 的等价操作。和同步屏障协调固定数目的线程不同，一个 phaser 能够协调不定数目的线程，这些线程可以在任何时候注册。为了实现这一功能，phaser 使用了 phase 和 phase 值。

第 7 章会涉及锁框架。

第7章

锁框架

java.util.concurrent.locks 包提供了一个包含多种接口和类的框架，它针对条件进行加锁和等待。不同于对象的内置加锁同步以及 java.lang.Object 的等待/通知机制，包含锁框架的并发工具类通过轮询锁、限时等待及其他方式改善了这种机制。

同步及低级别的锁

Java 支持同步以便线程能够安全地更新共享变量并且保证一条线程的更新对其他线程可见。你可以给方法或者代码块标记上 synchronized 关键字以达到同步的目的。这样的代码序列被称为临界区。Java 虚拟机（JVM）通过 JVM 指令 monitors、monitorenter 以及 monitorexit 来支持同步。

每一个 Java 对象都和一个监听器相关联，监听器是一个互斥（每次只允许一条线程在临界区中执行）构造，它阻止多条线程同时在临界区中并发执行。在线程可以进入临界区之前，它需要锁住监听器。如果这个监听器已经上锁，在监听器释放之前这条线程会一直阻塞（因为其他线程正在使用临界区）。

当线程在多核、多处理器的环境中锁住一个监听器，存储在主存中的共享变量的值会被读取到对应的拷贝中，然后存储在线程的工作内存（称为本

地内存或者缓存）。这一动作能够保证该线程使用这些变量最近的值并且不会污染这些值，我们称为可见性。线程会持续使用这些共享变量的拷贝。当离开临界区，线程会释放监听器，这些共享变量的值就被写回主存，以确保下一条线程进入临界区也能访问这些变量最近的值（volatile 关键字仅仅解决了可见性。）

锁框架包含了经常使用的锁、重入锁、条件、读写锁以及重入读写锁等类别，这些我都会在本章中进行探讨。同时，我也会简要地介绍 Java 8 中引入的 StampedLock 类。

7.1 锁

接口 Lock 提供了比监听器关联的锁更为弹性的锁操作，例如，当锁不可用时，可以立即退出对一个锁的请求。这个接口声明了如下方法：

- void lock()：获取锁。当锁不可用时，调用线程会被强制一直等待直到锁可用。
- void lockInterruptibly()：除非调用线程被中断，否则获得锁。当锁不可用时，调用线程被强制一直等待，直到锁可用或线程中断。中断会导致该方法抛出 java.lang.InterruptedException。
- Condition newCondition()：返回一个新的绑定到当前锁实例之上的 Condition 实例。当 Lock 的实现类不支持条件时，该方法抛出 java.lang.UnsupportedOperationException。

- boolean tryLock()：在该方法被调用时，锁可用则获得锁。当获得锁之后，该方法返回 true；反之返回 false。
- boolean tryLock(long time, TimeUnit unit)：当在指定等待时间内锁可用并且调用线程没被中断则获取锁，等待时间以 java.util.concurrent.TimeUnit 计（秒、毫秒等等）。当锁不可用时，调用线程会被强制一直等待，直到指定时间内锁可用或者线程中断。线程中断会导致该方法抛出 InterruptedException。当获得锁之后，该方法返回 true；反之返回 false。
- void unlock()：释放锁。

获取到的锁必须被释放。在同步方法和代码块的上下文中，隐式的监听锁和每个对象关联。所有的锁获取和释放都发生在块状结构中。当多个锁被获取，它们会以相反的顺序被释放，并且所有的锁都会在它们被获取的文本作用域中被释放掉。

在接口 Lock 实现的上下文中，锁的获取和释放可以更加弹性。举个例子，某些并发地遍历访问数据结构的算法需要使用“两手交替”或者“链状上锁”的方式：你获取节点 A 的锁，然后节点 B 的锁，然后释放 A 的锁并获取 C 的锁，之后又释放 B 的锁并获取 D 的锁等等。接口 Lock 的实现允许使用在不同的作用域中获取、释放锁这样的技术，同时以不同的顺序允许多个锁被获取和释放。

提升伸缩性伴随着额外的代价。块状锁的缺失移除了发生在同步方法和代码块上自动释放锁的功能。结果是，你通常应该遵循下列对于锁获取和释放的约定俗成：

```
Lock l = ...; // ... is a placeholder for code that obtains the lock
l.lock();
try
```

```
{
  // access the resource protected by this lock
}
catch (Exception ex)
{
  // restore invariants
}
finally
{
   l.unlock();
}
```

这一约定俗成能够保证获取到的锁总会被释放。

注意：所有 Lock 的实现都需要强制和内置的监听锁的内存同步语义保持一致。

7.2 重入锁

类 ReentrantLock 实现了接口 Lock，描述了一个可重入的互斥锁。这个锁和一个持有量相关联。当一条线程持有这个锁并且调用 lock()、lockUninterruptibly()或者任意一个 tryLock()方法重新获取锁，这个持有量就递增 1。当线程调用 unlock()方法，持有量就递减 1。当持有量降为 0，锁就会被释放掉。

ReentrantLock 提供了与通过同步方法、代码块得以访问的隐式监听锁同样的并发及内存语义。不过，它具备可扩展的功能并且在高线程争用的环境下（线程频繁地请求获取已经被其他线程持有的锁）具有更好的性能。这样，当

很多线程尝试访问共享资源时，JVM 会花费更少的时间来调度这些线程，而把更多的时间用于执行它们。

你可以通过调用下列构造函数之一初始化一个 ReentrantLock 的实例。

- ReentrantLock()：创建一个 ReentrantLock 的实例。该构造函数等价于 ReentrantLock(false)。
- ReentrantLock(boolean fair)：创建一个拥有指定公平策略的 ReentrantLock 的实例。当这个锁需要使用公平的排序策略时，给 fair 赋值为 true。在争用的环境下，这个锁倾向于将访问权限分配给等待最久的线程。

ReentrantLock 实现了 Lock 的方法。不过，它对 unlock()方法的实现在调用线程没有持有锁的情况下会抛出 java.lang.IllegalMonitorStateException。当然 ReentrantLock 也提供了自己的方法。举个例子，boolean isFair()方法返回公平策略，boolean isHeldByCurrentThread()方法在锁被当前线程持有的情况下，返回 true。清单 7-1 示范了 ReentrantLock。

清单 7-1 用重入锁获得同步功能

```
import java.util.concurrent.Executors;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.TimeUnit;

import java.util.concurrent.locks.ReentrantLock;

public class RLDemo
{
   public static void main(String[] args)
   {
```

```
ExecutorService executor = Executors.newFixedThreadPool(2);
final ReentrantLock lock = new ReentrantLock();

class Worker implements Runnable
{
   private final String name;

   Worker(String name)
   {
      this.name = name;
   }

   @Override
   public void run()
   {
      lock.lock();
      try
      {
         if (lock.isHeldByCurrentThread())
           System.out.printf("Thread %s entered critical section.%n",name);
         System.out.printf("Thread %s performing work.%n", name);
         try
         {
            Thread.sleep(2000);
         }
         catch (InterruptedException ie)
         {
            ie.printStackTrace();
```

```
            }
                System.out.printf("Thread %s finished working.%n", name);
            }
            finally
            {
                lock.unlock();
            }
        }
    }
    executor.execute(new Worker("ThdA"));
    executor.execute(new Worker("ThdB"));
    try
    {
        executor.awaitTermination(5, TimeUnit.SECONDS);
    }
    catch (InterruptedException ie)
    {
        ie.printStackTrace();
    }
    executor.shutdownNow();
  }
}
```

清单 7-1 的应用程序中，默认主线程创建了一对工作线程，它们进入临界区、在其中模拟执行任务，然后离开临界区。它们使用了 ReentrantLock 的 lock()和 unlock()方法来获取和释放一个重入锁。当一条线程调用 lock()方法而锁又不可用时，这个线程就会一直禁用（并且不能被调度），直到锁变为可用。

照下面编译清单 7-1：

```
javac RLDemo.java
```

运行程序：

```
java RLDemo
```

你应该能观察到类似下面的输出（消息的顺序可能略有出入）：

```
Thread ThdA entered critical section.
Thread ThdA performing work.
Thread ThdA finished working.
Thread ThdB entered critical section.
Thread ThdB performing work.
Thread ThdB finished working.
```

7.3 条件

接口 Condition 把 Object 的 wait 和 notification 方法（wait()，notify()和 notifyAll()）分解到不同的条件对象中。通过把这些条件和任意 Lock 实现的使用组合起来，起到让每个对象上具有多重等待集合的作用。这里 Lock 取代了同步方法、代码块，Condition 取代了 Object 的 wait、notificaiton 方法。

■ **注意**：一个 Condition 的实例原本就会绑定到一个锁上。可以使用 Lock 的 newCondition()方法去获取一个针对特定 Lock 实例的 Condition 实例。

Condition 声明了下列方法。

- `void await()`：在接收到信号或者被中断之前，强制调用线程一直等待。
- `boolean await(long time, TimeUnit unit)`：在接收到信号、被中断，或者超过指定的等待时间之前，强制调用线程一直等待。
- `long awaitNanos(long nanosTimeout)`：在接收到信号，被中断，或者超过指定的等待时间之前，强制当前线程一直等待。
- `void awaitUninterruptibly()`：在接收到信号之前，强制当前线程一直等待。
- `boolean awaitUntil(Date deadline)`：在接收到信号、被中断，或者超过指定的截止时间之前，强制当前的线程一直等待。
- `void signal()`：唤醒一个等待中的线程。
- `void signalAll()`：唤醒所有等待中的线程。

清单 7-2 用 Lock 和 Condition 获得同步功能

```
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class PC
{
   public static void main(String[] args)
   {
      Shared s = new Shared();
      new Producer(s).start();
```

```
      new Consumer(s).start();
   }
}

class Shared
{
   private char c;

   private volatile boolean available;

   private final Lock lock;

   private final Condition condition;

   Shared()
   {
      available = false;
      lock = new ReentrantLock();
      condition = lock.newCondition();
   }

   Lock getLock()
   {
      return lock;
   }

   char getSharedChar()
   {
```

```
    lock.lock();
    try
    {
       while (!available)
          try
          {
             condition.await();
          }
          catch (InterruptedException ie)
          {
             ie.printStackTrace();
          }
       available = false;
       condition.signal();
    }
    finally
    {
       lock.unlock();
       return c;
    }
}

void setSharedChar(char c)
{
    lock.lock();
    try
    {
       while (available)
```

```
            try
            {
               condition.await();
            }
            catch (InterruptedException ie)
            {
               ie.printStackTrace();
            }
         this.c = c;
         available = true;
         condition.signal();
      }
      finally
      {
         lock.unlock();
      }
   }
}
class Producer extends Thread
{
   private final Lock l;

   private final Shared s;

   Producer(Shared s)
   {
      this.s = s;
      l = s.getLock();
   }
```

```
   @Override
   public void run()
   {
      for (char ch = 'A'; ch <= 'Z'; ch++)
      {
         l.lock();
         s.setSharedChar(ch);
         System.out.println(ch + " produced by producer.");
         l.unlock();
      }
   }
}

class Consumer extends Thread
{
   private final Lock l;

   private final Shared s;

   Consumer(Shared s)
   {
      this.s = s;
      l = s.getLock();
   }

   @Override
   public void run()
```

```
    {
        char ch;
        do
        {
            l.lock();
            ch = s.getSharedChar();
            System.out.println(ch + " consumed by consumer.");
            l.unlock();
        }
        while (ch != 'Z');
    }
}
```

清单 7-2 和清单 3-2 的 PC 应用程序类似。不过，它用锁和条件替换了同步（synchronized）及等待/通知。

PC 的 main()方法初始化了类 Shared、Producer 以及 Consumer。Shared 实例被传递到 Producer 和 Consumer 的构造函数中，这些线程随后启动。

默认主线程调用 Producer 和 Consumer 构造函数。由于 Shared 实例也被 producer 和 consumer 线程所访问，该实例必须对这些线程可见（尤其是当这些线程运行在不同的核上时）。在每个 Producer 和 Consumer 中，笔者通过把 s 声明成 final 来达到这个目的。当然也可以将这一属性声明成 volatile，但是 volatile 暗示会对这一属性进行额外的写操作，事实上 s 在初始化之后就不会再改变了。

查看 Shared 的构造函数，注意到它通过 lock = new ReentrantLock(); 的方法创建了一个锁，并且通过 condition = lock.newCondition();创建了一个与之关联的条件。生产者和消费者可以通过 Lock getLock()方法访问这个锁。

当变量 available 为 true 时，生产者调用条件的 await()方法等待 available 变为 false。消费者在消费字符之后，发出信号给这个条件来唤醒生产者（因为可能存在假唤醒，而 available 还是 true，我使用循环代替了 if 语句。）

在离开循环之后，生产者线程记录下新的字符，将 available 赋值为 true 表明有一个新字符可被消费，同时发出信号给这个条件来唤醒一个等待中的消费者。最终，它释放了锁并且退出了 setSharedChar()方法。

注意：尽管字符是先产生后消费的，我还是给 Producer run()方法中 setSharedChar()/System.out.println()块以及 Consumer run()方法中的 getSharedChar()/System.out.println()块上了锁，以免应用程序在输出产生消息之前输出消费消息。

消费者线程的行为和 getSharedChar()方法，与我们之前对生产者线程的描述及 setSharedChar()方法类似。

注意：由于当前上下文不会抛出异常，我并没有使用 try/finally 这样的约定俗成来保证在 Producer 和 Consumer 的 run()方法里释放了锁。

照下面编译清单 7-2：

```
javac PC.java
```

运行程序：

```
java PC
```

你应该能观测到类似下面输出（这里只给出了输出的开始部分），这暗示了步调一致的同步（生产者线程在有东西被消费之前不会产生，而消费者线程在有东西被产生出来之前不会消费）。

```
A produced by producer.
A consumed by consumer.
B produced by producer.
B consumed by consumer.
C produced by producer.
C consumed by consumer.
D produced by producer.
D consumed by consumer.
```

7.4 读写锁

读写锁适用于对数据结构频繁读而较少修改的场景。举个例子，你可以创建了一个单词定义的在线辞典供多条线程并发读取，然而单条线程可能会不时地添加新的定义或更新已有的定义。锁框架针对这些场景提供了读—写锁机制，在读取时有更好的并发性，而写入时保证安全的互斥访问。这一机制基于接口 ReadWriteLock。

ReadWriteLock 维护了一对锁：一个锁针对只读操作，一个锁针对写操作。在没有写操作的时候，读锁可能会被多条读线程同时持有。写入锁是互斥的：只有单个线程可以修改共享数据（和 synchronized 关键字关联的锁也是互斥的）。

ReadWriteLock 声明了下列方法。

- Lock readLock()：返回用于读的锁。
- Lock writeLock()：返回用于写的锁。

7.5 重入读写锁

类 ReentrantReadWriteLock 实现了接口 ReadWriteLock，代表了一个与 ReentrantLock 具有相同语义的重入读—写锁。

你可以调用下列构造函数之一来初始化一个 ReentrantReadWriteLock 实例。

- ReentrantReadWriteLock()：创建一个 ReentrantReadWriteLock 的实例。该构造函数等价于 ReentrantReadWriteLock(false)。
- ReentrantReadWriteLock(boolean fair)：创建一个具有公平策略的 ReentrantReadWriteLock 的实例。当这个锁需要使用公平的顺序策略时，将 fair 赋值为 true。

注意：基于公平的顺序策略，若当前持有的锁被释放了，那要么是等待最久的单条写线程会被分配写锁，要么就是当一组读线程比所有等待中的写线程等待时间还长时，这组读线程会被分配读锁。

尝试获取一个公平读锁（非重入的）的线程会因为写锁被持有或者存在一条等待的写线程而被阻塞住。在等待最久的写线程已经获取并释放写锁之前，这条线程是没法获取到读锁的。如果等待中的写线程放弃了等待，致使一条或者多条读线程成为无写锁的队列当中最久的等待者，那些读线程就会被分配读锁。

除非读锁和写锁都没有被持有（暗示没有处于等待中的线程），否则尝试获取一个公平写锁（非重入的）的线程将会阻塞（非阻塞的 tryLock()方法并不遵循公平设置，它不管等待中的线程，而是尽可能立即获取锁）。

在初始化这个类之后，可以调用下列方法来获取读和写锁。

- `ReentrantReadWriteLock.ReadLock readLock()`：返回用于读的锁。
- `ReentrantReadWriteLock.WriteLock writeLock()`：返回用于写的锁。

每个内嵌的 `ReadLock` 和 `WriteLock` 类都实现了接口 `Lock` 并且实现了自己的方法。除此之外，`ReentrantReadWriteLock` 声明了像下面这对方法的额外方法。

- `int getReadHoldCount()`：返回被调用线程在这个锁上持有读锁的数量，当调用线程没有持有这个读锁，返回 0。读线程对每一个有上锁动作却没有释放锁动作的锁，都会有一个持有量。
- `int getWriteHoldCount()`：返回被调用线程在这个锁上持有写锁的数量，当调用线程没有持有这个写锁时，返回 0。写线程对每一个有上锁动作却没有释放锁动作的锁，都会有一个持有量。

为了演示 `ReadWriteLock` 和 `ReentrantReadWriteLock`，清单 7-3 展示了一个应用程序，其写线程产生单词定义的条目，而读线程持续随机地访问这些条目并打印出来。

清单 7-3　使用 ReadWriteLock 满足辞典程序的读、写线程

```
import java.util.HashMap;
import java.util.Map;

import java.util.concurrent.Executors;
import java.util.concurrent.ExecutorService;


import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReadWriteLock;
```

```
import java.util.concurrent.locks.ReentrantReadWriteLock;

public class Dictionary
{
   public static void main(String[] args)
   {
      final String[] words =
      {
         "hypocalcemia",
         "prolixity",
         "assiduous",
         "indefatigable",
         "castellan"
      };

      final String[] definitions =
      {
         "a deficiency of calcium in the blood",
         "unduly prolonged or drawn out",
         "showing great care, attention, and effort",

         "able to work or continue for a lengthy time without tiring",
         "the govenor or warden of a castle or fort"
      };

      final Map<String, String> dictionary = new HashMap<String, String>();

      ReadWriteLock rwl = new ReentrantReadWriteLock(true);
      final Lock rlock = rwl.readLock();
      final Lock wlock = rwl.writeLock();
```

```
Runnable writer = () ->
                  {
                     for (int i = 0; i < words.length; i++)
                     {
                        wlock.lock();
                        try
                        {
                           dictionary.put(words[i],
                                          definitions[i]);
                           System.out.println("writer storing " +
                                              words[i] + " entry");
                        }
                        finally
                        {
                           wlock.unlock();
                        }

                        try
                        {
                           Thread.sleep(1);
                        }
                        catch (InterruptedException ie)
                        {
                           System.err.println("writer " +
                                              "interrupted");
                        }
                     }
                  };
ExecutorService es = Executors.newFixedThreadPool(1);
es.submit(writer);
```

```
        Runnable reader = () ->
                          {
                            while (true)
                            {
                               rlock.lock();
                               try
                               {
                                  int i = (int) (Math.random() *
                                                 words.length);
                                  System.out.println("reader accessing " +
                                                     words[i] + ": " +
                                                     dictionary.get(words[i])
                                                     + " entry");
                               }
                               finally
                               {
                                  rlock.unlock();
                               }
                            }
                          };
        es = Executors.newFixedThreadPool(1);
        es.submit(reader);
    }
}
```

在清单 7-3 中，默认主线程首先创建单词和定义的字符串数组，由于它们得从匿名类中访问，因而定义成 **final**。创建了一个存储单词及其定义条目的 **map** 之后，线程获取一个重入的读写锁并开始访问读写锁。

针对写线程的 **runnable** 现在被创建出来，其 **run()**方法遍历了单词的数组，每次遍历都会锁住写锁。当此方法返回时，写线程就有了互斥的写锁，同

时能够通过调用 map 的 put 方法来更新 map。在输出一个标识添加单词的消息之后，写线程会释放锁，然后睡眠 1 毫秒给操作其他任务的线程出现的机会。接下来，基于线程池的 executor 会调用写线程的 runnable。

针对读线程的 runnable 后续会被创建出来。其 run()方法会重复地获取读锁，随机访问 map 中的条目，输出这个条目，然后释放读锁。接下来，从线程池中获取的 executor 会调用读线程的 runnable。

虽然我曾经因为不会抛出异常而没有遵循锁获取和释放的约定，但为了良好的格式，这里我还是指定了 try/fanally 结构。

照下面编译清单 7-3：

```
javac Dictionary.java
```

运行程序：

```
java Dictionary
```

正如我在执行时观测到的，你应该也能观测到如下输出，这里只给出了输出的开始部分（消息的顺序可能略有出入）：

```
writer storing hypocalcemia entry
writer storing prolixity entry
reader accessing hypocalcemia: a deficiency of calcium in the blood entry
writer storing assiduous entry
reader accessing assiduous: showing great care, attention, and effort entry
reader accessing castellan: null entry
reader accessing hypocalcemia: a deficiency of calcium in the blood entry
reader accessing assiduous: showing great care, attention, and effort entry
reader accessing indefatigable: null entry
reader accessing hypocalcemia: a deficiency of calcium in the blood entry
reader accessing hypocalcemia: a deficiency of calcium in the blood entry
reader accessing assiduous: showing great care, attention, and effort entry
reader accessing indefatigable: null entry
```

```
reader accessing prolixity: unduly prolonged or drawn out entry
reader accessing hypocalcemia: a deficiency of calcium in the blood entry
reader accessing castellan: null entry
reader accessing assiduous: showing great care, attention, and effort entry
reader accessing hypocalcemia: a deficiency of calcium in the blood entry
reader accessing prolixity: unduly prolonged or drawn out entry
reader accessing assiduous: showing great care, attention, and effort entry
reader accessing castellan: null entry
reader accessing hypocalcemia: a deficiency of calcium in the blood entry
reader accessing indefatigable: null entry
reader accessing castellan: null entry
reader accessing prolixity: unduly prolonged or drawn out entry
reader accessing hypocalcemia: a deficiency of calcium in the blood entry
writer storing indefatigable entry
reader accessing assiduous: showing great care, attention, and effort entry
reader accessing assiduous: showing great care, attention, and effort entry
```

警告：Java 8把StampedLock添加到java.util.concurrent.locks包中。根据其JDK 8的文档，StampedLock是一个具有3种控制读写访问模式的基于能力（capability-based）锁[①]。它用类似于ReentrantReadWriteLock的方式区分了互斥和非互斥的锁，而且还提供了ReentrantReadWriteLock锁不支持的乐观读。可以查看Heinz Kabutz博士关于Phaser and StampedLock并发同步器的视频演示（www.parleys.com/tutorial/5148922b0364bc17fc56ca4f/ chapter0/about）来学习StampedLock。当然，也可以看看这个演示的PDF文件（www.jfokus.se/jfokus13/preso/jf13_PhaserAndStampedLock.pdf）。

[①] 译者注：这里的capability指的是一种locking能力。

7.6 练习

下面的练习用来验证你对第 7 章内容的掌握程度。

1. 给出锁的定义。

2. 当线程进入临界区（通过 synchronized 保留字控制），Lock 对象与内置锁相比，最大的优势是什么？

3. 判断对错：当调用线程没有持有锁，ReentrantLock 的 unlock()方法会抛出 IllegalMonitorStateException。

4. 如何获取和特定 Lock 实例一起使用的一个 Condition 实例。

5. 判断对错：ReentrantReadWriteLock()创建了一个用于公平顺序策略的 ReentrantReadWriteLock 实例。

6. 给出 StampedLock 的定义。

7. 包 java.util.concurrent.locks 包含了类 LockSupport，这个 LockSupport 类的意义何在？

8. 用一个等价类替换下面的 ID 类，这个类使用 ReentrantLock 替代了 synchronized：

```
public class ID
{
   private static int counter; // initialized to 0 by default

   public static synchronized int getID()
   {
```

```
            int temp = counter + 1;
            try
            {
               Thread.sleep(1);
            }
            catch (InterruptedException ie)
            {
            }
            return counter = temp;
        }
    }
```

7.7　小结

包 java.util.concurrent.locks 提供了包含多种接口和类的框架一个，它针对条件进行加锁和等待。不同于对象的内置加锁同步以及 java.lang.Object 的等待/通知机制，包含锁框架的并发工具类通过轮询锁、限时等待及其他方式改善了这种机制。

锁框架包含了经常使用的锁、重入锁、条件、读写锁以及重入读写锁等类别，这些我都已经在本章中探讨过。同时，还简要地介绍了 Java 8 中引入的类 StampedLock。

第 8 章会涉及额外的并发工具。

第 8 章

额外的并发工具类

第 5 章到第 7 章介绍了并发框架，Executor（Callable 和 Future），同步器以及锁框架。本章中，我会通过介绍并发集合、原子变量、Fork/Join 框架以及 Completion Service 来覆盖所有的并发工具类。

注意：由于篇幅有限，我无法覆盖 Completable Future 的内容。如果你对这个话题感兴趣，我推荐你看看 Tomasz Nurkiewicz 这篇高质量的博客，题为“Java 8: Definitive Guide to CompletableFuture”，地址是 http://www.nurkiewicz.com/2013/05/java-8-definitive-guide-to.html。

8.1 并发集合

Java 的集合框架提供了位于 java.util 包下的诸多接口和类。其中接口包括了 List、Set 和 Map。类包含了 ArrayList、TreeSet 以及 HashMap。

ArrayList、TreeSet、HashMap 以及其他实现了这些接口的类都不是线程安全的。不过你可以使用类 java.util.Collections 中的同步包装方法让

它们变得安全。举个例子，可以向 Collections.synchronizedList()中传入一个 ArrayList 实例，以获得一个线程安全的 ArrayList 变体。

尽管它们对于在多线程环境中简化代码是必要的，但是线程安全的集合仍然有很多问题。

- 在遍历一个集合之前，获取锁是很有必要的。集合很可能在遍历过程当中被其他线程修改。如果一个锁并没有被获取并且集合被修改了，那很可能就会抛出java.util.ConcurrentModificationException。发生这种情况是因为集合框架的类返回了快速失败的迭代器，当集合在遍历过程中被修改时，这些迭代器就会抛出 ConcurrentModificationException。快速失败的迭代器对于并发应用程序经常是不方便的。
- 当同步的集合经常被多条线程频繁地访问，性能就成了问题。性能问题最终会影响应用程序的扩展能力。

并发工具类通过并发集合来应对这些问题。并发集合是具有并发性能和高扩展性、面向集合的类型，它们存储在 java.util.concurrent 包中。面向集合的类返回了弱一致性的迭代器，这类迭代器具有如下属性。

- 迭代开始之后，被删除但还没有通过迭代器的 next()方法被返回的元素，就不会再被返回了。
- 迭代开始之后被添加的元素可能会返回也可能不会返回。
- 在集合迭代的过程中，即便对集合做了改变，也没有任何元素会被返回超过一次。

下面列出了简短的面向并发的集合类型的样例，你可以在 java.util.concurrent 包中找到它们：

- BlockingQueue 是接口 java.util.Queue 的子接口，它支持阻塞操

作，即在获取一个元素之前，等待队列成为非空；在存储一个元素之前，等待队列中的空间变为可用。类 ArrayBlockingQueue、DelayQueue、LinkedBlockingQueue、PriorityBlockingQueue 以及 SynchronousQueue 都直接实现了这个接口。类 LinkedBlockingDeque 和 LinkedTransferQueue 通过实现 BlockingQueue 的子接口间接地实现了这个接口。

- ConcurrentMap 是 java.util.Map 的子接口，声明了额外的原子的方法 putIfAbsent()、remove()和 replace()。类 ConcurrentHashMap（等价于 java.util.HashMap 的并发版本）、类 ConcurrentNavigableMap 和 ConcurrentSkipListMap 都实现了这个接口。

Oracle 关于 BlockingQueue、ArrayBlockingQueue 以及其他面向并发集合类型的 Javadoc，都把这些类型标识成了集合框架的一部分。

8.1.1 使用 BlockingQueue 和 ArrayBlockingQueue

BlockingQueue 的 Javadoc 展示了生产者-消费者应用程序的核心，这比第 3 章中展示的同等的应用程序简单很多（请看清单 3-1），因为它不需要处理同步。清单 8-1 在更高级的生产者-消费者的等价程序中使用了 BlockingQueue 及其 ArrayBlockingQueue 实现。

清单 8-1 第 3 章 PC 应用程序阻塞队列的等价实现

```
import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
```

```
public class PC
{
    public static void main(String[] args)
    {
        final BlockingQueue<Character> bq;
        bq = new ArrayBlockingQueue<Character>(26);
        final ExecutorService executor = Executors.newFixedThreadPool(2);
        Runnable producer = () ->
                            {
                                for (char ch = 'A'; ch <= 'Z'; ch++)
                                {
                                    try
                                    {
                                        bq.put(ch);
                                        System.out.printf("%c produced by " +
                                                          "producer.%n", ch);
                                    }
                                    catch (InterruptedException ie)
                                    {
                                    }
                                }
                            };
        executor.execute(producer);
        Runnable consumer = () ->
                            {
                                char ch = '\0';
                                do
```

```
            {
                try
                {
                    ch = bq.take();
                    System.out.printf("%c consumed by " +
                                      "consumer.%n", ch);
                }
                catch (InterruptedException ie)
                {
                }
            }
            while (ch != 'Z');
            executor.shutdownNow();
        };
        executor.execute(consumer);
    }
}
```

清单 8-1 使用了 BlockingQueue 的 put()和 take()方法分别将一个对象放到阻塞队列中以及将一个对象从中移除。如果没有空间可以放置对象了，put()方法会被阻塞住，如果这个队列空了，take()方法也会阻塞住。

尽管 BlockingQueue 能够保证一个字符绝不会在它生产之前就被消费，但是应用程序的输出仍然可能相反。举个例子，如下是有一次运行中的部分输出：

```
Y consumed by consumer.
Y produced by producer.
Z consumed by consumer.
Z produced by producer.
```

第 3 章清单 3-2 中的 PC 应用通过在 setSharedChar()/System.out.

println()和 getSharedChar()/System.out.println()周围引入另外一层同步，克服了输出顺序的错误。第 7 章的清单 7-2 的 PC 应用中，通过将这些方法放置在 lock()/unlock()方法之间以克服同样的错误。

8.1.2 深入学习 ConcurrentHashMap

类 ConcurrentHashMap 和 HashMap 在行为上别无二致，但是设计成无需显式同步就可以工作于多线程的上下文。举个例子，你经常需要检查一个 map 中是否包含某个特定的值，当这个值不存在的时候，将它放进 map 中：

```
if (!map.containsKey("some string-based key"))
   map.put("some string-based key", "some string-based value");
```

尽管这段代码很简单并且似乎可以胜任此项工作，但是它却不是线程安全的。在调用 map.containsKey()和 map.put()方法之间，其他线程可能插入了这个条目，很可能会被覆盖掉。为了消除这个竞态条件，你必须显式地同步这段代码，示例如下：

```
synchronized(map)
{
   if (!map.containsKey("some string-based key"))
      map.put("some string-based key", "some string-based value");
}
```

这个方法的问题在于，当检查键值是否存在并且如果不存在便将条目添加到 map 时，你把整个 map 的读写操作都给锁住了。当多条线程同时访问这个 map 时，这样的锁会影响性能。

泛型的 ConcurrentHashMap<V>类通过提供 V putIfAbsent(K key, V value)方法来应对这个问题，当键值不存在时，它会把 key/value 条目放进 map 中。这个方法等价于下面的代码段，但具有更好的性能：

```
synchronized(map)
{
   if (!map.containsKey(key))
      return map.put(key, value);
   else
      return map.get(key);
}
```

使用 putIfAbsent()方法，原来的代码段就可以翻译成如下简单的代码段了：

```
map.putIfAbsent("some string-based key", "some string-based value");
```

注意：Java 8 添加了超过 30 个新方法，改善了 ConcurrentHashMap，通过聚合操作大量地支持了 lambda 表达式和 Streams API。进行聚合操作的方法包括 forEach()方法（forEach()、forEachKey()、forEachValue()以及 forEachEntry()）、查找方法（search()、searchKeys()、searchValues()以及 searchEntries()），以及化约方法（reduce()、reduceToDouble()、reduceToLong()等），其他方法（如 mappingCount()和 newKeySet()）也被加入其中。由于 JDK 8 的改变，ConcurrentHashMaps（以及从中构建的类）现在作为缓存更好用。改善缓存的变化反映在为键计算值的方法上，对扫描（可能会逃逸）条目的改良以及对拥有大量元素的 map 更好的支持上。

8.2 原子变量

和对象监听器关联的那些内置锁一直以来都有性能不佳的问题。尽管性能

已经有所提高，但是它们依旧是创建 Web 服务器以及其他在高争用环境下要求更好的扩展及性能之类的应用的瓶颈。

大量研究探寻在同步上下文中创建非阻塞的算法，这些算法可以从根本上提高性能。当多条线程争用同样的数据时，线程不会阻塞，所以这些算法提高了扩展性。当然，线程也不会遭遇死锁和其他活跃性问题。

Java 5 通过引入包 `java.util.concurrent.atomic` 提供了创建高效非阻塞的算法。根据这个包的 JDK 文档，`java.util.concurrent.atomic` 提供了小型工具类，支持在单个变量上进行无锁及线程安全的操作。

包 `java.util.concurrent.atomic` 中的类把 `volatitle` 标注的数值、属性以及数组元素扩展成也能支持原子的、有条件的更新，这样外部的同步就不再需要了。换句话说，你可以在无需外部同步的情况下获取和 `volatile` 变量相关联的内存互斥语义。

注意：术语 atomic 和 indivisible 一般被认为是相同的，尽管原子是可以分割的。

包 `java.util.concurrent.atomic` 中一些类描述如下：

- `AtomicBoolean`：可被原子更新的布尔值。
- `AtomicInteger`：可被原子更新的整型值。
- `AtomicIntegerArray`：一个整型的数组，其元素可被原子更新。
- `AtomicLong`：可被原子更新的长整型值。
- `AtomicLongArray`：一个长整型的数组，其元素可被原子更新。
- `AtomicReference`：可被原子更新的对象引用。
- `AtomicReferenceArray`：一个对象引用的数组，其元素可被原子更新。

原子变量用于实现计数器、序列生成器（例如 java.util.concurrent.ThreadLocalRandom）以及其他构造。在线程高争用的环境中，这些构造要求互斥而不影响性能。举个例子，清单 8-2 中的 ID 类，其类方法 getNextID() 返回唯一的长整型标识符。

清单 8-2 借助 synchronized，以线程安全的方式返回唯一的标识符

```
class ID
{
   private static volatile long nextID = 1;

   static synchronized long getNextID()
   {
      return nextID++;
   }
}
```

尽管这段代码确实被同步过（可见性也被保证了），但是在线程高争用的环境下，synchronized 关键字关联的内置锁会影响性能。甚至，诸如死锁之类的活跃性问题也会发生。清单 8-3 展示了如何用一个原子变量替换 synchronized 的方式来避免这些问题。

清单 8-3 借助 AtomicLong，以线程安全的方式返回唯一的标识符

```
import java.util.concurrent.atomic.AtomicLong;

class ID
{
   private static AtomicLong nextID = new AtomicLong(1);

   static long getNextID()
   {
```

```
        return nextID.getAndIncrement();
    }
}
```

在清单 8-3 中，我将 nextID 从 long 型转换成一个 AtomicLong 的实例，并初始化成 1。同时，将 getNextID()方法重构成了对 AtomicLong 的 getAndIncrement()方法的调用，该方法会在一个不可分割的步骤里把 AtomicLong 实例的内部长整型递增 1 并返回之前的值。此时并不存在显式的同步。

警告：包 java.util.concurrent.atomic 包含了 DoubleAccumulator、DoubleAdder、LongAccumulator 以及 LongAdder 类，这些类解决了从多线程中维护单个计数、求和以及更新某些值的上下文中的扩展性问题。这些新类“从内部利用了争用-递减的技术，相较于原子变量带来更大吞吐量的提升。这就使得在大多数应用程序中放松原子性保证成为一种可接受的可能。”

8.2.1　理解原子魔法

Java 低级的同步机制，强制使用互斥（持有锁的线程保证一组变量被互斥地访问）以及可见性（被保护变量的更改对后续获取锁的其他线程可见），以如下的方式影响了硬件的使用和扩展能力。

- 争用的同步（多条线程持续竞争一个锁）是非常昂贵的，并且结果是吞吐量很糟糕。这种代价主要是由频繁的上下文切换引起的（把中央处理单元从一条线程切换到另外一条）。每次上下文切换的操作，都会花费多个处理器周期来完成。相反地，现代 Java 虚拟机（JVM）使得非争用的同步变得廉价起来。
- 当持有锁的线程被延迟了（举个例子，由于调度延迟），所有需要那个锁的线程都没法继续运行，硬件就得不到充分利用。

尽管你可能认为 volatile 是同步的备选项，但事实并非如此。volatile 变量只能解决可见性的问题，它们无法应用于安全地实现原子的读-改-写的序列。这个原子的序列对于实现线程安全的计数器以及其他需要互斥访问的实体是有必要的。不过，并发工具集（例如类 java.util.concurrent.Semaphore）提供了一个负责性能提升的备选项，它就是 compare-and-swap。

Compare-and-swap(CAS)是一个针对非抢占式微处理器的一条指定指令的宽泛术语，这条指令读取内存的位置，比较读到的值和期望的值，当读到的值和期望的值匹配时，就将新值存储到该内存位置；否则，什么事也不会发生。现代微处理器提供了多种 CAS 的变体。例如，Intel 微处理器提供了 cmpxchg 家族的指令，而旧的 PowerPC 微处理器提供了等价的 load-link（如 lwarx）以及 store-conditional（如 stwcx）指令。

CAS 支持原子的读-改-写序列。你通常会像下面这样使用 CAS：

（1）从地址 A 读出值 x。

（2）在 x 上进行一个多步计算，衍生出一个新的值 y。

（3）使用 CAS 把 A 的值从 x 改成 y。当操作这些步骤时，如果 A 的值没有发生改变，CAS 就成功了。

为了理解 CAS 的好处，以清单 8-2 中的 ID 类为例，它返回一个唯一的标识符。由于这个类将 getNextID()方法声明成了 synchronized，高争用环境下的监听锁会导致过多的上下文切换，这样会阻碍所有的线程并且导致应用程序无法很好得扩展。

假设存在一个 CAS 类，它存储了基于整型值的 value。更进一步，它提供了原子的方法 int getValue()以返回值，以及 int compareAndSwap(int expectedValue, int newValue)以实现 CAS（此场景的背后，CAS 依赖于 Java 本地接口来访问特定微处理器的 CAS 指令）。

这个 compareAndSwap()方法以原子的方式执行下列指令的序列：

```
int readValue = value;                  // Obtain the stored value.
if (readValue == expectedValue)         // If stored value not modified ...
   value = newValue;                    // ... change to new value.
return readValue;                       // Return value before a potential change.
```

清单 8-4 展示了新版的 ID 类，使用了 CAS 类以一种高性能的方式获取唯一的标识（忘掉使用 JNI 导致的性能后果，假设我们可以直接访问特定微处理器的 CAS 指令）。

清单 8-4　通过 CAS，以一种线程安全的方式返回唯一的 ID

```
class ID
{
   private static CAS value = new CAS(1);

   static long getNextID()
   {
      int curValue = value.getValue();
      while (value.compareAndSwap(curValue, curValue + 1) != curValue)
         curValue = value.getValue();
      return curValue - 1;
   }
}
```

ID 类封装了一个被实例化成整型值 1 的 CAS 实例，并且声明了一个 getNextID()方法用于获取当前的标识符，之后从该实例中递增这个值。在获取该实例的当前值之后，getNextID()会反复地调用 compareAndSwap()方法，直到 curValue 的值不再改变（被其他线程）。这个方法在返回前面的值之后就可以自由地改变 curValue 的值了。当不涉及锁或高争用环境下不会有过多的上下文切换时，性能就得以提高，并且代码也会更具扩展性。

作为 CAS 如何改善并发工具的例子，考虑 java.util.concurrent.locks.ReentrantLock。这个类在高线程争用的环境下较 synchronized 提供了更好的性能。为了提升性能，抽象类 java.util.concurrent.locks.AbstractQueuedSynchronizer 的子类管理了 ReentrantLock 的同步方式。这个类利用了未经文档化的类 sun.misc.Unsafe 及其 compareAnd SwapInt() 的 CAS 方法。

原子变量的类也利用了 CAS。甚至，它们提供的方法具有下列形式：

```
boolean compareAndSet(expectedValue, updateValue)
```

这个方法（在不同的类中参数的类型不同）在变量当前等于 expectedValue 时，会原子性地给它设置 updateValue，成功返回 true。

8.3 Fork/Join 框架

让代码执行得更快一些总归是必要的。过去，这一需求总是通过提升微处理器的速度或者提供多个处理器得以满足。不过，2003 前后，微处理器的速度由于自然限制停止了增长。为了补偿，处理器制造商开始往他们的处理器中添加多个处理内核，借助大量的并行来提升速度。

注意：并行指的是从多个处理器和核心的一些组合中同时运行线程。相反地，并发则是一种更为宽泛形式的并行，并发中，线程可以同时运行或者通过上下文切换看上去像是同时运行，也就是虚拟并行。一些人更进一步将并发描述成程序或操作系统的一个属性，而将并行看作同时运行多条线程的运行时行为。

Java 提供底层的线程特性以及线程池之类的高级并发工具类来支持并发。但是，并发的问题在于它无法最大化利用可用的处理器以及核心的资源。举个例子，假设你创建了一种排序算法将一个数组分成两个部分，安排两条线程分别排序每个部分，在线程都完成之后合并结果。

假设每条线程在不同的处理器上运行。由于不同数量的元素会在数组的每部分上重新排序，很可能一条线程先于其他线程完成但在合并发生前必须等待。在这一场景下，一个处理器资源就被浪费掉了。

这个问题（和相关的代码冗余难以阅读的问题）可以通过递归地把任务拆分成子任务然后组合结果的方式解决。这些任务会并行执行，并且几乎在同一时间完成（如果不在同一时刻），它们的结果会被合并、传递到上一层子任务的栈上。这样几乎没有任何处理器时间会浪费在等待上，并且这些递归的代码也更简洁、（大部分）更容易理解。Java 提供了 Fork/Join 框架来实现这一场景。

Fork/Join 框架由特定的 ExecutorService 和线程池构成。ExecutorService 可以运行任务，并且这个任务会被分解成较小的任务，它们从线程池中被 fork（被不同的线程执行）出来，在 join（即它的所有子任务都完成了）之前会一直等待。

Fork/Join 使用了任务窃取来最小化线程的争用和开销。线程池中的每条工作线程都有自己的双端工作队列并且会将新任务放到这个队列中。它从队列的头部读取任务。如果队列是空的，工作线程就尝试从另一个队列的末尾获取一个任务。窃取操作不会很频繁，因为工作线程会采用后进先出（LIFO）的顺序将任务放入它们的队列中，同时工作项的规模会随着问题分割成子问题而变小。你一开始把任务交给一个中心的工作线程，之后它会持续将这个任务分解成更小的任务。最终所有的工作线程都只会涉及很少量的同步操作。

Fork/Join 框架绝大部分由 java.util.concurrent 包中的 ForkJoinPool、ForkJoinTask、ForkJoinWorkerThread、RecursiveAction、RecursiveTask 以及 CountedCompleter 类组成。

- ForkJoinPool 是一个针对运行中的 ForkJoinTasks 的 java.util.concurrent.ExecutorService 实现。ForkJoinPool 的实例提供了一个来自非 ForkJoinTask 客户端的提交的入口，同时提供了管理和监听操作。
- ForkJoinTask 是一个抽象的基类，专门针对那些运行在 ForkJoinPool 上下文中的任务。一个 ForkJoinTask 的实例是一个类似线程的实体，不过比正常的线程轻量许多。大量的任务以及子任务可能会被少量 ForkJoinPool 中的真实线程以某些使用限制作为代价进行托管。
- ForkJoinWorkerThread 描述了被 ForkJoinPool 管理的一个线程，它执行 ForkJoinTask。
- RecursiveAction 描述了一个递归、无结果的 ForkJoinTask。
- RecursiveTask 描述了一个递归、有结果的 ForkJoinTask。
- CountedCompleter 描述了一个带有完成动作（完成一个 fork/join 任务）的 ForkJoinTask，在被触发且没有其他滞留的动作时这个任务会被执行。

Java 文档提供基于 RecursiveAction 的任务（例如排序）以及基于 RecursiveTask 的任务(例如计算 Fibonacci 数列)。你也可以使用 RecursiveAction 去完成矩阵乘法（请看 http://en.wikipedia.org/wiki/Matrix_multiplication)。举个例子，假设你已经创建了清单 8-5 中的 Matrix 类来表示由指定行和列数组成的矩阵。

清单 8-5 表示矩阵的类

```
public class Matrix
{
   private final int[][] matrix;

   public Matrix(int nrows, int ncols)
   {
      matrix = new int[nrows][ncols];
   }

   public int getCols()
   {
      return matrix[0].length;
   }

   public int getRows()
   {
      return matrix.length;
   }

   public int getValue(int row, int col)
   {
      return matrix[row][col];
   }

   public void setValue(int row, int col, int value)
   {
      matrix[row][col] = value;
```

```
    }
}
```

清单 8-6 示范了以一种单线程的方式进行两个矩阵的乘法。

清单 8-6　由标准的矩阵乘法算法来计算两个矩阵的相乘

```
public class MatMult
{
   public static void main(String[] args)
   {
      Matrix a = new Matrix(1, 3);
      a.setValue(0, 0, 1); // | 1 2 3 |
      a.setValue(0, 1, 2);
      a.setValue(0, 2, 3);
      dump(a);
      Matrix b = new Matrix(3, 2);
      b.setValue(0, 0, 4); // | 4 7 |
      b.setValue(1, 0, 5); // | 5 8 |
      b.setValue(2, 0, 6); // | 6 9 |
      b.setValue(0, 1, 7);
      b.setValue(1, 1, 8);
      b.setValue(2, 1, 9);
      dump(b);
      dump(multiply(a, b));
   }

   public static void dump(Matrix m)
   {
      for (int i = 0; i < m.getRows(); i++)
      {
```

```
            for (int j = 0; j < m.getCols(); j++)
               System.out.printf("%d ", m.getValue(i, j));
            System.out.println();
         }
         System.out.println();
      }

      public static Matrix multiply(Matrix a, Matrix b)
      {
         if (a.getCols() != b.getRows())
            throw new IllegalArgumentException("rows/columns mismatch");
         Matrix result = new Matrix(a.getRows(), b.getCols());
         for (int i = 0; i < a.getRows(); i++)
            for (int j = 0; j < b.getCols(); j++)
               for (int k = 0; k < a.getCols(); k++)
                  result.setValue(i, j, result.getValue(i, j) +
                                        a.getValue(i, k) * b.getValue(k, j));
         return result;
      }
   }
```

清单 8-6 中的 MatMult 类声明了一个 multiply()方法表示矩阵的乘法。在验证了矩阵算法的基本条件，即第一个矩阵（a）的列数等于第二个矩阵（b）的行数之后，multiply()创建了一个结果矩阵，并进入一系列的内嵌循环来进行乘法操作。

这些循环的实际工作如下：对于 a 中每一行，将行中的每一个值和 a 中对应列的每一个值相乘。然后将乘法的结果相加，同时把整个结果存储到 a 中行索引（i）和 b 中列索引（j）指定的位置上。

编译清单 8-6 和清单 8-5，它们必须在同一个目录，如下所示：

```
javac MultMat.java
```

运行程序：

```
java MultMat
```

你应该能观测到如下的输出，即 1 行 3 列的矩阵乘以 3 行 2 列的矩阵得到一个 1 行 2 列的矩阵：

```
1 2 3


4 7
5 8
6 9


32 50
```

计算机科学家将这类的算法分类为 O(n*n*n)，读作“大 O n 的立方”或者“近似 n 的立方”。这种表示法是一种分类算法性能的抽象方式（不会陷入诸如微处理器速度这样的细节当中）。一个 O(n*n*n)分类暗示着很差的性能，并且随矩阵的大小成倍增长，性能变得更差。

可以通过把每个行乘列的乘法任务分配到独立的类似线程的实体，以提升性能。清单 8-7 展示了如何在 Fork/Join 框架的上下文中完成这一场景。

清单 8-7　借助 Fork/Join 框架将两个矩阵的实例相乘

```
import java.util.ArrayList;
import java.util.List;


import java.util.concurrent.ForkJoinPool;
```

```
import java.util.concurrent.RecursiveAction;

public class MatMult extends RecursiveAction
{
   private final Matrix a, b, c;
   private final int row;

   public MatMult(Matrix a, Matrix b, Matrix c)
   {
      this(a, b, c, -1);
   }

   public MatMult(Matrix a, Matrix b, Matrix c, int row)
   {
      if (a.getCols() != b.getRows())
         throw new IllegalArgumentException("rows/columns mismatch");
      this.a = a;
      this.b = b;
      this.c = c;
      this.row = row;
   }

   @Override
   public void compute()
   {
      if (row == -1)
      {
         List<MatMult> tasks = new ArrayList<>();
```

```
            for (int row = 0; row < a.getRows(); row++)
               tasks.add(new MatMult(a, b, c, row));
            invokeAll(tasks);
         }
         else
            multiplyRowByColumn(a, b, c, row);
      }

      public static void multiplyRowByColumn(Matrix a, Matrix b, Matrix c,
                                             int row)
      {
         for (int j = 0; j < b.getCols(); j++)
            for (int k = 0; k < a.getCols(); k++)
               c.setValue(row, j, c.getValue(row, j) +
                     a.getValue(row, k) * b.getValue(k, j));
      }

      public static void dump(Matrix m)
      {
         for (int i = 0; i < m.getRows(); i++)
         {
            for (int j = 0; j < m.getCols(); j++)
               System.out.print(m.getValue(i, j) + " ");
            System.out.println();
         }
         System.out.println();
      }
```

```
    public static void main(String[] args)
    {
        Matrix a = new Matrix(2, 3);
        a.setValue(0, 0, 1); // | 1 2 3 |
        a.setValue(0, 1, 2); // | 4 5 6 |
        a.setValue(0, 2, 3);
        a.setValue(1, 0, 4);
        a.setValue(1, 1, 5);
        a.setValue(1, 2, 6);
        dump(a);
        Matrix b = new Matrix(3, 2);
        b.setValue(0, 0, 7); // | 7 1 |
        b.setValue(1, 0, 8); // | 8 2 |
        b.setValue(2, 0, 9); // | 9 3 |
        b.setValue(0, 1, 1);
        b.setValue(1, 1, 2);
        b.setValue(2, 1, 3);
        dump(b);
        Matrix c = new Matrix(2, 2);
        ForkJoinPool pool = new ForkJoinPool();
        pool.invoke(new MatMult(a, b, c));
        dump(c);
    }
}
```

清单 8-7 中的 MatMult 类继承自 RecursiveAction。为了完成有意义的工作，RecursiveAction 的 void compute()方法被重写了。

注意：尽管 compute()方法通常会将一个任务递归地分割成多个子任务，但是我选它来做乘法任务（为了言简意赅）确实稍有不同。

创建矩阵 a 和 b 之后，清单 8-7 中的 main()方法创建了矩阵 c，同时初始化了 ForkJoinPool。之后它初始化了 MatMult，将这 3 个 Matrix 的实例作为参数传递到 MatMult(Matrix a, Matrix b, Matrix c)构造函数中，并且调用 ForkJoinPool 的 T invoke(ForkJoinTask< T > task)方法开始运行初始化任务。这个方法一直不会返回，直到初始化任务和其所有子任务都已经完成。

当 compute()最开始被调用的时候（row 等于-1），它创建了一个 MatMult 任务的列表并且把这个列表传递到了 RecursiveAction 的 Collection<T> invokeAll(Collection<T> tasks)方法（从 ForkJoinTask 中继承）。这个方法 fork 了所有列表中的任务，并开始执行它们。之后会一直等待，直到 invokeAll()方法返回（这也会 join 全部任务），也即 boolean isDone()方法（也是从 ForkJoinTask 中继承的）对于每个任务都返回 true。

注意这个 task.add(new MatMult(a, b, c, row));方法的调用。这一调用将指定的 row 值赋给了一个 MatMult 的实例。当 invokeAll()方法被调用了，每个任务的 compute()方法会被调用并且检测到 row 被赋予不同的值（不同于-1）。之后它会针对指定的 row 执行 multiplyRowByColumn(a, b, c, row);方法。

编译清单 8-7（javac MatMult.java），运行最终程序（java MatMult）。你应该能观测到如下的输出：

```
1 2 3
4 5 6
```

```
7 1
8 2
9 3

50 14
122 32
```

8.4 Completion Service

一个 completion service 就是 java.util.concurrent.CompletionService<V> 接口的实现，用于从已完成任务（消费者）结果的消费中解耦新的异步任务（生产者）的生产。V 是这个任务返回结果的类型。

生产者通过调用某个 submit()方法提交一个任务执行（通过一个工作线程）：其中有个方法接收一个 callable 参数，而另一个方法接收一个 runnable 参数和一个任务完成的返回值。每个方法都会返回一个 Future<V>的实例，代表该任务的未完成状态。之后你可以调用 poll()方法来轮训这个任务的完成状态或者调用这个阻塞式的 take()方法。

消费者通过调用 take()方法获取一个已完成的任务。这个方法在任务已经完成之前会一直阻塞。之后它会返回一个 Future<V>对象来表示这一完成的任务，调用 Future<V>的 get()方法即可获取结果。

连同 CompletionService<V>一起，Java 7 引入了 java.util.concurrent.ExecutorCompletionService<V>类，这个类提供 executor 用于支持任务执

行。它保证当提交过的任务完成时，这些任务会被放到一个可被 take()方法访问的队列中。

为了演示 CompletionService 和 ExecutorCompletionService，我会重新做一遍第 5 章给出的计算欧拉数的程序。清单 8-8 中的源代码表示一个新的应用程序，它提交了两个 callable 任务以不同的精度计算数字。

清单 8-8　借助 Completion Service 计算欧拉数

```
import java.math.BigDecimal;
import java.math.MathContext;
import java.math.RoundingMode;

import java.util.concurrent.Callable;
import java.util.concurrent.CompletionService;
import java.util.concurrent.ExecutorCompletionService;
import java.util.concurrent.Executors;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Future;

public class CSDemo
{
   public static void main(String[] args) throws Exception
   {
      ExecutorService es = Executors.newFixedThreadPool(10);
      CompletionService<BigDecimal> cs =
         new ExecutorCompletionService<BigDecimal>(es);
      cs.submit(new CalculateE(17));
      cs.submit(new CalculateE(170));
      Future<BigDecimal> result = cs.take();
```

```
            System.out.println(result.get());
            System.out.println();
            result = cs.take();
            System.out.println(result.get());
            es.shutdown();
        }
    }

    class CalculateE implements Callable<BigDecimal>
    {
        final int lastIter;

        public CalculateE(int lastIter)
        {
            this.lastIter = lastIter;
        }

        @Override
        public BigDecimal call()
        {
            MathContext mc = new MathContext(100, RoundingMode.HALF_UP);
            BigDecimal result = BigDecimal.ZERO;
            for (int i = 0; i <= lastIter; i++)
            {
                BigDecimal factorial = factorial(new BigDecimal(i));
                BigDecimal res = BigDecimal.ONE.divide(factorial, mc);
                result = result.add(res);
            }
```

```
        return result;
    }

    private BigDecimal factorial(BigDecimal n)
    {
        if (n.equals(BigDecimal.ZERO))
            return BigDecimal.ONE;
        else
            return n.multiply(factorial(n.subtract(BigDecimal.ONE)));
    }
}
```

清单 8-8 展示了两个类：CSDemo 和 CalculateE。CSDemo 驱动程序，而 CalculateE 则描述了欧拉数的计算任务。

CSDemo 的 main()方法首先创建了一个 executor service，以供后续执行任务。之后创建了一个 completion service 用于完成任务。两个计算任务随后被提交到了这个 completion service，每个计算任务都会异步地执行。对于每个任务，调用这个 completion service 的 take()方法会返回任务的 future 对象，future 的 get()方法用于获取任务的结果以供后续输出。

CalculateE 包含的代码和第 5 章中展示的代码几乎相同（请看清单 5-1）。唯一的不同在于，将一个 LASTITER 常量改成了一个 lastIter 变量，此变量用于记录最后一次执行的迭代（以及决定这个数字小数部分的精度）。

照下面编译清单 8-8：

```
javac CSDemo.java
```

运行程序：

```
java CSDemo
```

你应该能观测到如下的输出：

```
2.71828182845904507051604779584860506117897963525103269890073500406522504250
484331405588797434424574173003945406271l
```

```
2.71828182845904523536028747135266249775724709369995957496696762772407663035
354759457138217852516642746389611628165412481304872986538030830542556283824 5
913460032675144581911560494210526286856488476919630428470349167770684812212 6
664838550045128841929851772268853216753574895628940347880297133296754744949 3
758350055422838463145284198638405011249720440692822554843276680620741498059 3
297816148195171199144814650 6
```

注意：如果你想知道 executor service 和 completion service 之间的区别，可以这样考虑，使用 executor service，写完提交任务的代码之后，你需要写一些代码去高效地获取任务的结果；使用 completion service，这个工作就更加自动化了。另一种方式就是比较它们的构造，executor service 为任务提供了一个输入队列和多条工作线程，而 completion service 则为任务提供了输入队列、工作线程以及一个输出队列来存储任务的结果。

8.5 练习

下面的练习用来验证你对第 8 章内容的掌握程度。

1. 说明线程安全集合的两个问题。
2. 给出并发集合的定义。

3. 什么是一个弱一致性的迭代器？

4. 描述 BlockingQueue 接口。

5. 描述 ConcurrentMap 接口。

6. 描述 ArrayBlockingQueue、LinkedBlockingQueue 和 BlockingQueue 实现类。

7. 判断对错：面向并发的集合类型是集合框架的一部分。

8. 描述 ConcurrentHashMap 类。

9. 使用 ConcurrentHashMap，你该如何检查是否一个 map 包含了特定的值，并且当这个值不存在的时候，无需外部同步，把这个值放到 map 中。

10. 给出原子变量的定义。

11. 类 AtomicIntegerArray 描述了什么？

12. 判断对错：volatile 支持原子的读-改-写序列。

13. 通过使用并发工具类能够改善了哪些方面的性能？

14. 描述 Fork/Join 框架。

15. 确认构成 Fork/Join 框架的主要类型。

16. 为了通过 RecursiveAction 完成有意义的工作，你应该覆写哪个方法？

17. 给出 completion service 的定义。

18. 如何使用一个 completion service？

19. 如何通过一个 completion service 来运行任务？

20. 将下列的表达式转换成它们对应的原子变量等价物：

```
int total = ++counter;
int total = counter--;
```

8.6　小结

本章通过介绍并发集合、原子变量、Fork/Join 框架以及 completion service 来覆盖所有的并发工具类。

存储在 `java.util.concurrent` 包中的并发集合是具有并发性能和高扩展性面向集合的类型，它们克服了 `ConcurrentModificationException` 以及线程安全的集合带来的性能问题。

原子变量是封装了单个变量，并且支持在单个变量上进行无锁及线程安全的操作的实例，如 `AtomicInteger`。

Fork/Join 框架由特定的 executor service 和线程池构成。executor service 可以运行任务，并且这个任务会被分解成较小的任务，它们从线程池中被 fork（被不同的线程执行）出来，在 join（即它的所有子任务都完成了）之前会一直等待。

一个 `completion service` 就是 `java.util.concurrent.CompletionService<V>`接口的实现，用于从已完成任务（消费者）结果的消费中解耦新的异步任务（生产者）的生产。`V` 是这个任务返回结果的类型。

附录 A 会给出每章的练习题的答案。

第 3 部分
附录

附录 A

练习题答案

从第 1 章到第 8 章，每一章都是以习题作为结尾的，这些习题旨在测试你对每章内容的理解程度。本附录给出这些习题的答案。

第 1 章　Thread 和 Runnable

1. 线程就是一条在程序代码中独立执行的路径。

2. 一个 runnable 就是一段封装在对象中的代码序列，它的类实现了 Runnable 接口。

3. 类 Thread 提供了一个底层操作系统线程架构的统一接口。Runnable 接口为关联了 Thread 对象的线程提供执行代码。

4. 我们有两种方式去创建一个 Runnable 对象，创建一个实现了 Runnable 接口的匿名类或者使用 lambda 表达式。

5. 我们有两种方式将一个 runnable 关联到一个 Thread 对象上，把一个 runnable 传递到 Thread 类的接收 runnable 参数的构造函数里，或者在构造函数不接收 runnable 参数的时候，继承 Thread 类继而后重写它的 void run()

方法。由于 Thread 实现了 Runnable 接口，所以 Thread 对象也是 runnables。

6. 构成 Thread 状态的 5 个标识分别是线程名称、线程存活的标识、线程的执行状态（是否可执行？）、线程的优先级以及该线程是否为守护线程的标识。

7. 答案是错。默认的线程名称以 Thread-前缀开头。

8. 通过调用接收线程名称的 Thread 构造函数或者 void setName(String name)方法，你可以给线程一个非默认的名称。

9. 通过调用 Thread 的 boolean isAlive()方法判断一条线程是否活着。

10. Thread.State 枚举的常量包括 NEW（该状态下线程还没有开始执行），RUNNABLE（该状态下线程正在 JVM 中执行），BLOCKED（该状态下线程被阻塞，正在等待一个监听锁），WAITING（该状态下线程无限制地等待另外一个线程执行特定的操作）以及 TERMINATED（该状态下线程已经退出）。

11. 你可以通过调用 Thread 的 Thread.State getState()方法获取当前的线程执行状态。

12. 优先级指的是线程的相对重要性。

13. 使用 setPriority()会影响应用程序在操作系统之间的可移植性，因为不同的调度器会采取不同的方式来处理优先级。

14. 传递给优先级的值介于 Thread.MIN_PRIORITY 和 Thread.MAX_PRIORITY 之间。

15. 答案是对。一个守护线程会在应用程序最后一个非守护线程消失之后自动死亡，因此应用程序才能终止。

16. 如果这个线程之前已经启动运行或者已经死亡，这个方法就会抛出 java.lang.IllegalThreadStateException。

17. 为了终止无限运行的应用程序，可以在 Windows 或者非 Windows 系统上同时按住 Ctrl 和 C 键。

18. 组成 Thread 中断机制的方法包括 void interrupt()、static boolean interrupted()以及 boolean isInterrupted()。

19. 答案是错。boolean isInterrupted()方法并不会清除线程的中断状态。这个中断状态并不会受到影响。

20. 当线程被中断时，它会抛出 InterruptedException。

21. 一个忙循环就是设计来消耗时间的一条循环语句。

22. Thread 中让一条线程等待另外一条线程死亡的方法有 void join()、void join(long millis)和 void join(long millis, int nanos)。

23. Thread 中让一条线程睡眠的方法有 void sleep(long millis)和 void sleep(long millis, int nanos)。

24. 清单 A-1 展示了第 1 章中的 IntSleep 应用程序。

清单 A-1 中断一条睡眠中的后台线程

```
public class IntSleep
{
   public static void main(String[] args)
   {
      Runnable r = new Runnable()
                   {
                      @Override
                      public void run()
                      {
                         while (true)
                         {
                            System.out.println("hello");
                            try
                            {
```

```
                    Thread.sleep(100);
                }
                catch (InterruptedException ie)
                {
                    System.out.println("interrupted");
                    break;
                }
            }
        }
    };
Thread t = new Thread(r);
t.start();
try
{
    Thread.sleep(2000);
}
catch (InterruptedException ie)
{
}
t.interrupt();
  }
}
```

第2章 同步

1. 线程中的3个问题分别是竞态条件、数据竞争以及缓存变量。

2．答案是错。当计算的正确性取决于相对时间或者调度器所控制的多线程交叉时，竞态条件就会发生。

3．同步指的是一种JVM的特性，它用于保证两条及两条以上的线程不会同时在一个临界区中执行。

4．同步的两种属性分别是互斥和可见性。

5．同步是基于监听器实现出来的，它是控制对临界区进行访问的并发构造，必须不可分割地执行。每个Java对象都关联着一个监听器，线程可以通过上锁或解锁的方式获取或者释放监听器上的锁。

6．答案是对。已经获取锁的线程在调用`Thread`的任意`sleep()`方法时是不会释放锁的。

7．你可以通过在方法头前面包含一个`synchronized`关键字的方式指定一个同步方法。

8．你可以通过`synchronized(object) {}`这种语法的方式指定一个同步块。

9．活跃性这个词代表着某件正确的事情最终会发生。

10．3种活跃性的挑战分别是死锁、活锁以及饿死（也称为无限延迟）。

11．`volatile`关键字有别于`synchronized`关键字之处在于`volatitle`只处理了可见性问题，而`synchronized`却处理了互斥和可见性问题。

12．答案是对。Java可以在无需同步的情况下安全地访问一个`final`的属性。

13．类`CheckingAccount`的线程问题是`withdraw()`方法中`if (amount <= balance)`和`balance -= amount;`（这会导致取到的钱超出可用余额）存在的竞态条件以及`balance`属性可能被缓存的问题。这个`balance`属性会被缓存在多处理器/多核系统中并且缓存的拷贝会被`withdrawal`线程使用，这个

拷贝可能和当初通过默认主线程在构造函数中的设置的初始 balance 不一致。

14. 清单 A-2 展示在第 2 章被调用的 CheckingAccount 应用程序。

清单 A-2 修复存在问题的账户检查程序

```
public class CheckingAccount
{
   private volatile int balance;

   public CheckingAccount(int initialBalance)
   {
      balance = initialBalance;
   }

   public synchronized boolean withdraw(int amount)
   {
      if (amount <= balance)
      {
         try
         {
            Thread.sleep((int) (Math.random() * 200));
         }
         catch (InterruptedException ie)
         {
         }
         balance -= amount;
         return true;
      }
      return false;
```

```
    }

    public static void main(String[] args)
    {
        final CheckingAccount ca = new CheckingAccount(100);
        Runnable r = new Runnable()
                     {
                        @Override
                        public void run()
                        {
                           String name = Thread.currentThread().getName();
                           for (int i = 0; i < 10; i++)
                               System.out.println (name + " withdraws $10: " +
                                                   ca.withdraw(10));
                        }
                     };
        Thread thdHusband = new Thread(r);
        thdHusband.setName("Husband");
        Thread thdWife = new Thread(r);
        thdWife.setName("Wife");
        thdHusband.start();
        thdWife.start();
    }
}
```

这段程序使用了 volatitle 关键字应对潜在的缓存问题，并用 synchronized 关键字满足了互斥的需要。

第3章 等待和通知

1. 所谓的条件（condition）指的就是持续执行的前置条件。

2. 支持条件的 API 是由 Object 的 3 个 wait()方法、一个 notify()及一个 notifyAll()方法组成的。这些 wait()方法等待某个条件成立，而 notify()和 notifyAll()方法则在这个条件成立时通知处于等待中的线程。

3. 答案是对。wait()方法是可以被中断的。

4. 你可以调用 notifyAll()方法唤醒在同一对象监听器上等待的所有线程。

5. 答案是错。已经获取锁的线程在调用 Object 上的任意 wait()方法时会释放锁。

6. 条件队列就是一种数据结构，用于存储那些等待某个条件成立的线程。这些等待中的线程称为等待集合。

7. 当你在同步上下文之外调用此 API 的任意方法时，会导致 IllegalMonitorStateException 发生。

8. 假唤醒指的是线程在没有被通知、被中断或者超时的情况下醒来。

9. 你应该在一个循环中调用 wait()方法以保证活跃性和安全。

10. 清单 A-3 展示了第 3 章中的 Await 应用程序。

清单 A-3 使用 wait()以及 notifyAll()来创建一个更高级的并发构造

```
public class Await
{
```

```
    static volatile int count;

    public static void main(String[] args)
    {
       Runnable r = () ->
                    {
                       Thread curThread = Thread.currentThread();
                       System.out.printf("%s has entered runnable and is " +
                                         "waiting%n", curThread.getName());
                       synchronized(Await.class)
                       {
                          count++;
                          try
                          {
                             Thread.sleep(2000);
                             while (count < 3)
                                Await.class.wait();
                          }
                          catch (InterruptedException ie)
                          {
                          }
                       }
                       System.out.printf("%s has woken up and is " +
                                         "terminating%n",
                                         curThread.getName());
                    };
       Thread thdA = new Thread(r, "thdA");
       Thread thdB = new Thread(r, "thdB");
```

```
        Thread thdC = new Thread(r, "thdC");
        thdA.start();
        thdB.start();
        thdC.start();
        r = new Runnable()
            {
                @Override
                public void run()
                {
                    try
                    {
                        while (count < 3)
                            Thread.sleep(100);
                        synchronized(Await.class)
                        {
                            Await.class.notifyAll();
                        }
                    }
                    catch (InterruptedException ie)
                    {
                    }
                }
            };
        Thread thd = new Thread(r);
        thd.start();
    }
}
```

第 4 章 额外的线程能力

1. 线程组就是一组线程，由类 ThreadGroup 表示。

2. 使用一个线程组，你可以对其中的所有线程进行统一操作以简化线程管理。

3. 因为多数有用的 ThreadGroup 方法都已经废弃，并且在获取一组活跃线程数和列举这些线程之间存在“检查时间到使用时间”这一类别的竞态条件，所以你应该避免使用线程组。

4. 由于 ThreadGroup 在处理线程执行过程中产生的异常方面做出的贡献，你还是应该对它有所了解。

5. 每个 ThreadLocal 的实例代表了一个线程局部变量，它为访问该变量的每个线程提供了单独的存储槽。

6. 答案是对。如果当线程调用 get()方法时，值不存在，那么它会调用 initialValue()方法。

7. 你可以使用类 InheritableThreadLocal 把一个值从父线程传到子线程当中。

8. 类 Timer 和 TimerTask 组成了 Timer 框架。

9. 答案是错。Timer()会创建一个新的定时器，其任务执行线程会以非守护线程的方式运行。

10. 在固定延迟执行中，每次执行都是相对于上次执行的实际发生时间的。当某次执行因为一些原因被延迟了（比如垃圾回收），后续的执行也都会被延迟。

11. 你可以调用 schedule()方法来调度一个固定延迟执行的任务。

12. 在以固定速率执行的方式当中，每次执行都是相对于初次执行的调度时间而被调度的。当某个执行因为一些原因被延迟了（比如垃圾回收），两次及两次以上的执行将会迅速相继发生。

13. Timer 的 cancel()和 TimerTask 的 cancel()方法的区别在于，Timer 的 cancel()会终止定时器并放弃当前任意被调度的定时任务，而 TimerTask 的 cancel()方法仅仅会取消正在调用的定时任务。

14. 清单 A-4 展示了第 4 章中的 BackAndForth 应用程序。

清单 A-4 借助 Timer，反复地将一个星号前后移动

```
import java.util.Timer;
import java.util.TimerTask;

public class BackAndForth
{
   static enum Direction { FORWARDS, BACKWARDS }

   public static void main(String[] args)
   {
      TimerTask task = new TimerTask()
                       {
                          final static int MAXSTEPS = 20;

                          volatile Direction direction = Direction.FORWARDS;

                          volatile int steps = 0;
```

```
                @Override
                public void run()
                {
                    switch (direction)
                    {
                        case FORWARDS : System.out.print("\b ");
                                        System.out.print("*");
                                        break;
                        case BACKWARDS: System.out.print("\b ");
                                        System.out.print("\b\b*");
                    }

                    if (++steps == MAXSTEPS)
                    {
                        direction =
                            (direction == Direction.FORWARDS)
                            ? Direction.BACKWARDS
                            : Direction.FORWARDS;

                        steps = 0;
                    }
                }
            };
        Timer timer = new Timer();
        timer.schedule(task, 0, 100);
    }
}
```

第 5 章　并发工具类和 Executor 框架

1. 并发工具类是一组用于克服 Java 底层线程问题的类和接口的框架。特别地，低级的并发原语如 synchronized 和 wait()/notify()经常难以正确使用。过于依赖这些并发原语会导致性能问题，进而影响了应用程序的扩展性。Java 底层的线程功能也并未包含线程池以及信号量这样的高级构造。

2. 包 java.util.concurrent、java.util.concurrent.atomic 以及 java.util.concurrent.locks 构成了并发实用工具类。

3. 一个任务是这样一种对象，它的类实现了 java.lang.Runnable（一个可运行的任务）或者 java.util.concurrent.Callable 接口（一个可被调用的任务）。

4. 一个 executor 就是这样一种对象，它的类直接或间接地实现了 java.util.concurrent.Executor 接口，这样便从任务执行机制中解耦任务的提交操作。

5. Executor 接口仅仅只关注 Runnable 接口。也就是说，一个可运行任务是没法简单地向其调用者返回结果值的（因为 Runnable 的 run()方法不会返回结果值）；Executor 接口也没有提供一种方式来追踪正在执行中的任务进程、取消正在执行的任务或者确定运行的任务什么时候结束执行；Executor 无法执行一组可运行的任务；而且 Executor 也没有为应用程序提供一种关闭 executor 的方式（更为正确地关闭 executor）。

6. ExecutorService 接口克服了 Executor 的局限性。

7. Runnable 的 run()方法和 Callable 的 call()方法之间的区别如下：run()无法返回结果值，而 call()方法可以；run()方法无法抛出受检异常，而 call()方法可以。

8. 答案是错。你可以从 Callable 的 call()方法中抛出受检和非受检异常，但是只能从 Runnable 的 run()方法中抛出非受检的异常。

9. future 是这样一种对象，它的类实现了 Future 接口。它代表了一种异步的计算并且提供了取消任务、返回任务结果值以及确定任务是否已经结束的方法。

10. 类 Executors 的 newFixedThreadPool()方法创建了一个线程池复用固定数量的线程操作一个共享的无限队列。至多有 nThreads 个线程同时处理任务。如果额外的任务在所有线程都活跃时被提交了，它们会在队列中等待一条可用线程。如果在 executor 停止之前，任何线程因为执行中的失败而终止，而后续任务的又需要执行的时候，一条新的线程会取代其位置。这些在线程池中的线程会一直存在，直到 executor 被显式地关闭。

11. 清单 A-5 展示了第 5 章中的 CountingThreads 应用程序。

清单 A-5 基于 Executor 的线程计数

```
import java.util.concurrent.Executors;
import java.util.concurrent.ExecutorService;

public class CountingThreads
{
   public static void main(String[] args)
   {
      Runnable r = new Runnable()
                   {
```

```
            @Override
            public void run()
            {
                String name = Thread.currentThread().getName();
                int count = 0;
                while (true)
                    System.out.println(name + ": " + count++);
            }
        };
        ExecutorService es = Executors.newFixedThreadPool(2);
        es.submit(r);
        es.submit(r);
    }
}
```

12. 清单A-6展示了第5章中的CountingThreads应用程序，线程的名称是自定义的：

清单A-6　基于Executor的A、B线程计数

```
import java.util.concurrent.Executors;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.ThreadFactory;

public class CountingThreads
{
   public static void main(String[] args)
   {
      Runnable r = new Runnable()
                   {
```

```
                @Override
                public void run()
                {
                   String name = Thread.currentThread().getName();
                   int count = 0;
                   while (true)
                      System.out.println(name + ": " + count++);
                }
             };
      ExecutorService es =
         Executors.newSingleThreadExecutor(new NamedThread("A"));
      es.submit(r);
      es = Executors.newSingleThreadExecutor(new NamedThread("B"));
      es.submit(r);
   }
}

class NamedThread implements ThreadFactory
{
   private volatile String name; // newThread() could be called by a
                                 // different thread

   NamedThread(String name)
   {
      this.name = name;
   }

   @Override
```

```
    public Thread newThread(Runnable r)
    {
       return new Thread(r, name);
    }
}
```

第6章 同步器

1. 同步器指的是控制通用同步方法的类。

2. 倒计时门闩会导致一条或多条线程在“门口”一直等待直到另一条线程打开这扇门，线程才得以继续运行。它是由一个计数变量和两个操作组成的，这两个操作分别是“导致一条线程等待直到计数变为0”以及“递减计数变量”。

3. 调用 CountDownLatch 的 void countDown()方法，当计数降至0时释放所有等待线程。

4. 同步屏障允许一组线程彼此互相等待，直到抵达某个公共的屏障点。因为该屏障在等待线程被释放之后可以重用，所以称它为可循环使用的屏障。该同步器对于这类数量固定，并且互相之间必须不时等待彼此的多线程应用很有用。

5. 答案是错。当有线程正在等待时该同步屏障被重置了（通过 reset()方法）以及当同步屏障在 await()方法被调用或任意线程正在等待时同步屏障被打破，CyclicBarrier 的 int await()方法就会抛出 java.util.concurrent.BrokenBarrierException。

6. 交换器提供了一个线程彼此之间能够交换对象的同步点。每条线程都会往这个交换器的 exchange()方法传入一些对象，匹配伙伴线程，同时接收伙伴线程中的对象作为返回值。

7. Exchanger 的 V exchange(V x)会在这个交互点上等待其他线程到达（除非调用线程被中断了），之后将所给对象传入其中，接收其他线程的对象作为返回。

8. 信号量维护了一组许可证（permit），以约束访问被限制资源的线程数。当没有可用的许可证时，线程的获取尝试会一直阻塞，直到其他的线程释放一个许可证。

9. 信号量有两种类型，分别是计数信号量（当前值可以被递增加 1）和二进制信号量或互斥信号量（当前值只能取 0 或 1）。

10. phaser 是一个更加弹性的同步屏障。和同步屏障一样，一个 phaser 使得一组线程在屏障上等待，在最后一条线程到达之后，这些线程得以继续执行。phaser 也提供 barrierAction 的等价操作。和同步屏障协调固定数目的线程不同，一个 phaser 能够协调不定数目的线程，这些线程可以在任何时候注册。为了实现这一功能，phaser 使用了 phase 和 phase 值。

11. 类 Phaser 的 int register()方法返回用于分类抵达的 phase 值。如果这个值是负数，该 phaser 就会终止，此时注册不产生任何影响。这个数值称为抵达 phase 值。

12. 清单 A-7 展示了在第 6 章中调用的 PC 应用程序。

清单 A-7　基于信号量的生产者和消费者

```
import java.util.concurrent.Semaphore;

public class PC
```

```
{
   public static void main(String[] args)
   {
      Shared s = new Shared();
      Semaphore semCon = new Semaphore(0);
      Semaphore semPro = new Semaphore(1);
      new Producer(s, semPro, semCon).start();
      new Consumer(s, semPro, semCon).start();
   }
}

class Shared
{
   private char c;

   void setSharedChar(char c)
   {
      this.c = c;
   }

   char getSharedChar()
   {
      return c;
   }
}

class Producer extends Thread
{
   private final Shared s;
```

```
    private final Semaphore semPro, semCon;

    Producer(Shared s, Semaphore semPro, Semaphore semCon)
    {
        this.s = s;
        this.semPro = semPro;
        this.semCon = semCon;
    }

    @Override
    public void run()
    {
        for (char ch = 'A'; ch <= 'Z'; ch++)
        {
            try
            {
                semPro.acquire();
            }
            catch (InterruptedException ie)
            {
            }
            s.setSharedChar(ch);
            System.out.println(ch + " produced by producer.");
            semCon.release();
        }
    }
}
class Consumer extends Thread
{
```

```
    private final Shared s;
    private final Semaphore semPro, semCon;

    Consumer(Shared s, Semaphore semPro, Semaphore semCon)
    {
        this.s = s;
        this.semPro = semPro;
        this.semCon = semCon;
    }

    @Override
    public void run()
    {
        char ch;
        do
        {
            try
            {
                semCon.acquire();
            }
            catch (InterruptedException ie)
            {
            }
            ch = s.getSharedChar();
            System.out.println(ch + " consumed by consumer.");
            semPro.release();
        }
        while (ch != 'Z');
    }
}
```

第 7 章 锁框架

1. 锁就是实现了接口 Lock 的类的实例，它提供了比 sychronized 保留字更为弹性的锁操作。锁也能通过 Condition 对象支持等待/通知机制。

2. 当线程进入临界区时，较之内置锁，Lock 对象最大的好处就是具备退出请求锁的能力。

3. 答案是对。当调用线程未持有锁时，ReentrantLock 的 unlock()方法就会抛出。

4. 你可以调用 Lock 的 Condition newCondition()方法获取与特定 Lock 实例相关联的 Condition 实例。

5. 答案是错。ReentrantReadWriteLock()方法创建了一个不带有公平顺序策略的 ReentrantReadWriteLock。

6. JDK 8 引入的 StampedLock 是一种基于能力（capability-based）的锁，拥有 3 种控制读写访问的模式。它用类似于 ReentrantReadWriteLock 的方式区分了互斥和非互斥的锁，而且还提供了 ReentrantReadWriteLock 锁不支持的乐观读。

7. LockSupport 的目的在于为创建锁以及其他同步类提供基本的线程阻塞原语。

8. 清单 A-8 展示了第 7 章中的 ID 类。

清单 A-8 基于 ReentrantLock 的 ID 生成器

```
import java.util.concurrent.locks.ReentrantLock;
```

```
public class ID
{
   private static int counter; // initialized to 0 by default

   private final static ReentrantLock lock = new ReentrantLock();

   public static int getID()
   {
      lock.lock();
      try
      {
         int temp = counter + 1;
         try
         {
            Thread.sleep(1);
         }
         catch (InterruptedException ie)
         {
         }
         return counter = temp;
      }
      finally
      {
         lock.unlock();
      }
   }
}
```

第 8 章 额外的并发工具类

1. 线程安全集合存在的两大问题是可能抛出 ConcurrentModificationException 以及糟糕的性能。因为在迭代集合的过程中，可能会有其他的线程修改集合，所以在迭代之前获取一个锁是很有必要的。如果没有获取锁并且集合遭到修改，那很可能引发 ConcurrentModificationException。同样地，如果同步集合被多条线程频繁访问，性能就会很糟糕。

2. 并发集合是具有并发性能和高扩展性面向集合的类型，它们位于 java.util.concurrent 包中。

3. 弱一致性的迭代器具有以下属性：

- 迭代开始之后，被删除但还没有通过迭代器的 next()方法被返回的元素，就不会再被返回了。
- 迭代开始之后被添加的元素可能会返回也可能不会返回。
- 在集合迭代的过程中，即便对集合做了改变，也没有任何元素会被返回超过一次。

4. BlockingQueue 是接口 java.util.Queue 的子接口，它支持阻塞操作，即在获取一个元素之前，等待队列成为非空；在存储一个元素之前，等待队列中的空间变为可用。

5. ConcurrentMap 是 java.util.Map 的子接口，声明了额外的原子的方法 putIfAbsent()、remove()和 replace()。

6. ArrayBlockingQueue 是一个基于数组的有界阻塞队列。LinkedBlockingQueue 是一个基于链接节点的非有界阻塞队列。

7. 答案是对。面向并发的集合类型是集合框架的一部分。

8. 类 ConcurrentHashMap 和 HashMap 在行为上别无二致，但是设计成无需显式同步就可以工作于多线程的上下文。

9. 使用 ConcurrentHashMap 时，你会调用其 putIfAbsent()方法去检查一个 map 是否包含指定的值。当这个值不存在的时候，不依赖外部的同步将它放进此 map 中。

10. 原子变量封装了单个变量，并且支持在这个变量上进行无锁和线程安全的操作，如 AtomicInteger。

11. 类 AtomicIntegerArray 描述了一个整型的数组，其元素可被原子更新。

12. 答案是错。volatile 并不支持原子的读—改—写的操作序列。

13. compare-and-swap 指令负责并发工具类承诺的性能提升。

14. Fork/Join 框架由特定的 executor service 和线程池构成。executor service 可以运行任务，并且这个任务会被分解成较小的任务，它们从线程池中被 fork（被不同的线程执行）出来，在 join（即它的所有子任务都完成了）之前会一直等待。

15. Fork/Join 框架绝大部分由 java.util.concurrent 包中的 ForkJoinPool、ForkJoinTask、ForkJoinWorkerThread、RecursiveAction、RecursiveTask 以及 CountedCompleter 类组成。

16. 为了完成有意义的工作，RecursiveAction 的 void compute()方法需要重写。

17. 一个 completion service 就是 java.util.concurrent.CompletionService 接口的实现，用于从已完成任务（消费者）结果的消费中解耦新的异步任务（生产者）的生产。V 是这个任务返回结果的类型。

18. 你可以照下面的方式使用 completion 服务：通过调用 CompletionService 任意的 submit()方法提交一个用于执行（通过工作线程）的任务。每个方法都会返回一个 Future<V>实例代表该任务的未完成状态。之后你可以调用 poll()方法来轮训这个任务的完成状态或者调用阻塞式的 take()方法。消费者通过调用 take()方法获取一个已完成的任务。这个方法在任务已经完成之前会一直阻塞。之后它会返回一个 Future<V>对象来表示这一完成的任务，调用 Future<V>的 get()方法即可获取结果。

19. 你可以通过 completion 服务并借助类 ExecutorCompletionService<V>来执行任务，这个类实现了 CompletionService<V>接口，同时提供 executor 用于支持任务执行。

20. 和 int total = ++counter;等价的原子变量如下：

```
AtomicInteger counter = new AtomicInteger(0);
int total = counter.incrementAndGet();
```

和 int total = counter--;等价的原子变量如下：

```
AtomicInteger counter = new AtomicInteger(0);
int total = counter.getAndDecrement();
```

附录 B

Swing 中的线程

Swing 是一个平台独立、基于模型-视图-控制器的 GUI 工具集，它被用来创建 Java 应用程序的图形界面。在本附录中，我首先会探索 Swing 的线程架构，之后会探索 Swing 的 API，这些 API 用于防止在图形上下文中使用额外线程导致的问题。最后，我给出一个基于 Swing 的幻灯片应用程序作为本附录中的重要例子，同时用这种好玩的方式结束本书。

注意： 我会假设你对 Swing 的 API 及其应用程序的架构有一定的经验。

B.1 单线程编程模型

Swing 遵循单线程编程模型。它设计成单线程而非多线程的原因在于设计多线程图形工具集的经验已经表明多线程必然会导致死锁和竞态条件。如果想了解更多相关问题，可以查看“Why are GUIs Single-threaded？”博客（http://codeidol.com/java/java-concurrency/ GUI-Applications/Why-are-GUIs-Single-threaded/）。

用于渲染图像和处理事件的线程被称为事件——分派（event-dispatch）线程（EDT）。EDT 处理来自底层抽象窗口工具集的事件队列中事件，同时调用 GUI 组件（例如按钮）的事件监听器，这些监听器在此线程上处理事件。组件甚至会在 EDT 上重绘它们自身（响应 `paint()`方法调用时会导致 `paintComponent()`、`paintBorder()`、`paintChildren()`方法被调用）。

一定得小心关注你的代码如何同 EDT 交互，确保 Swing 程序能正确工作。有两条原则需要记住：

- 总是在 EDT 上创建 Swing GUI。
- 绝对不要延迟 EDT。

单线程的 Swing 导致的一个结果就是你只能在 EDT 上创建 Swing 应用程序的 GUI。在其他线程中创建 GUI 是不对的，包括用于运行 Java 应用程序中 `main()`方法的默认主线程。

大部分 Swing 对象（如 `javax.swing.JFrame` 对象，描述了 GUI 中拥有菜单条和边界的顶级“框架”窗口）都不是线程安全的。从多条线程中访问这些对象，会招致线程干涉的风险或者内存不一致的错误：

- 线程干涉：当两条线程作用在相同的数据上，执行了不同的操作。举个例子，一条线程在另一条线程更新基于长整型的计数变量时，读取此变量。由于长整型在 32 位的机器上读取和写入操作需要两次读写访问，很可能读线程读取了该变量部分当前值，然后写线程更新了这个变量，再然后读线程读完了该变量的剩余部分。结果就是读线程读到了不正确的值。
- 内存不一致错误：运作在不同的处理器或者核心的两条及其以上线程拥有相同数据的非一致视图。举个例子，某个处理器或者核心上的一条写线程更新了一个计数变量，之后另一个处理器或者核心上的读线

程读取了该变量。不过，由于存在用于提升性能的缓存机制，线程不会去访问主存当中单一的变量拷贝，取而代之地，每条线程都会从它们自己的本地存储（缓存）中访问这个变量的拷贝。

如果 GUI 不是在 EDT 之上创建的会发生什么问题呢？John Zukowski 在 JavaWorld 上发表的《Swing threading and the event-dispatch thread》一文中给出了一个场景（`www.javaworld.com/article/2077754/core-java/swing-threading-and-the-event-dispatch-thread.html`）。

Zukowski 展示了一个例子，往 frame 窗口容器组件中添加一个容器的监听器。当组件从 frame 当中添加或者移除时，监听器的方法就会被调用。他演示了默认主线程意识到 frame 窗口之前，EDT 在监听器方法中已经开始运行代码。

注意：能被感知到意味着一个组件的 `paint()`方法要么已经被调用，要么可能被调用。frame 窗口中 `setVisible(true)`、`show()`和 `pack()`中任意一个方法在该容器组件上被调用，这个 frame 窗口就被感知到了。在 frame 窗口被感知之后，它包含的所有组件也都被感知到了。另外一种感知组件的方式就是把它添加到已经被感知的容器当中。

EDT 在一个监听器方法中开始运行之后，如果默认主线程继续初始化 GUI，组件可能会被默认主线程创建并被 EDT 访问。EDT 可能试图在这些组件存在之前访问它们。这样做可能会使应用程序崩溃。

即便在 EDT 从监听器方法中访问这些组件之前，默认主线程创建了它们，EDT 还是可能看到不一致的视图（由于缓存）而无法访问指向新组件的引用。应用程序崩溃（可能抛出 `java.lang.NullPointerException` 对象）十有八九会发生。

清单 B-1 展示了 `ViewPage` 的源码，即一个浏览 Web 页面 HTML 的 Swing 应用程序。这个应用程序就遭受了以上两个问题。

清单 B-1　一个有问题的 Swing 应用，用于浏览 Web 页面 HTML

```
import java.awt.BorderLayout;
import java.awt.Dimension;
import java.awt.EventQueue;

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import java.io.InputStream;
import java.io.IOException;

import java.net.URL;

import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;
import javax.swing.JTextField;

public class ViewPage
{
   public static void main(String[] args)
   {
      final JFrame frame = new JFrame("View Page");
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      JPanel panel = new JPanel();
      panel.add(new JLabel("Enter URL"));
```

```
    final JTextField txtURL = new JTextField(40);
    panel.add(txtURL);
    frame.getContentPane().add(panel, BorderLayout.NORTH);
    final JTextArea txtHTML = new JTextArea(10, 40);
    frame.getContentPane().add(new JScrollPane (txtHTML),
                               BorderLayout.CENTER);
    ActionListener al = (ae) ->
{
    InputStream is = null;
    try
    {
       URL url = new URL(txtURL.getText());
       is = url.openStream();
       StringBuilder sb = new StringBuilder();
       int b;
       while ((b = is.read()) != -1)
          sb.append((char) b);
       txtHTML.setText(sb.toString());
    }
    catch (IOException ioe)
    {
       txtHTML.setText(ioe.getMessage());
    }
    finally
    {
       txtHTML.setCaretPosition(0);
       if (is != null)
          try
```

```
                {
                    is.close();
                }
                catch (IOException ioe)
                {
                }
            }
        };
        txtURL.addActionListener(al);
        frame.pack();
        frame.setVisible(true);
    }
}
```

清单 B-1 中的 `main()`方法创建了一个 GUI，它由一个用于输入 Web 页面 URL 的文本框以及一个用于显示页面 HTML 的可滚动文本区域组成。在输入 URL 之后按下回车键，触发 `ViewPage` 获取并显示 HTML。

照下面编译清单 B-1：

```
javac ViewPage.java
```

运行程序：

```
java ViewPage
```

你应该能观测到如图 B-1 展示的 GUI（由一个样例 URL 和部分结果 Web 页面的 HTML 构成）。

该应用程序的第一个问题是 GUI 在默认的主线程而非 EDT 被创建出来。尽管再运行 `ViewPage` 时你可能不会遇到问题，但是依然会有潜在的线程干涉以及内存不一致问题。

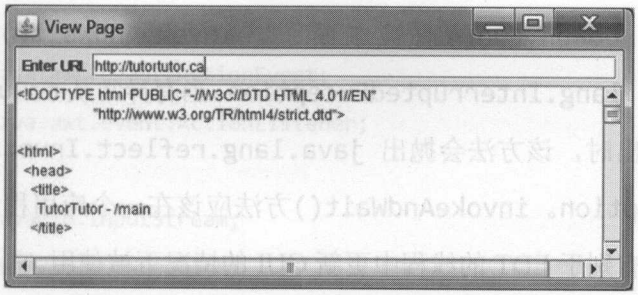


图 B-1 文本框输入一个 URL 并且在可滚动文本区域浏览 Web 页面输出

第二个问题是那个运行动作监听器的 EDT，它用于响应文本框之上按下回车的操作。针对这个 URL 打开一个 InputStream 及读取其内容到一个 StringBuilder 中的代码会推迟 EDT。GUI 在此期间不会响应。

B.2 线程化 API

Swing 提供了一系列的 API 用于克服前面提及的 EDT 的问题。本节中，我会介绍这些 API。同时，我也会介绍 Swing 版本的定时器，它和第 4 章中展示的定时器框架有很大不同。

B.2.1 SwingUtilities 和 EventQueue

类 `javax.swing.SwingUtilities` 提供了一组在 Swing 上下文中很有用的静态方法。其中有 3 个对于使用 EDT 和避免提及的问题很管用。

- `void invokeAndWait(Runnable doRun)`：使得 `doRun.run()`方法在 EDT 中同步地执行。这个调用块会一直阻塞直到所有的等待事件都

已经被处理完，之后 doRun.run()方法返回。当该方法在等待 EDT 执行完 doRun.run()时被中断了，invokeAndWait()方法会抛出 java.lang.InterruptedException。当有异常从 doRun.run()方法中抛出时，该方法会抛出 java.lang.reflect.InvocationTargetException。invokeAndWait()方法应该在一个应用程序线程需要从其他有别于 EDT 的线程中更新 GUI 的情况下被使用。它不应该从 EDT 中被调用。

- void invokeLater(Runnable doRun)：在所有等待事件都已经被处理完之后，使得 doRun.run()方法将在 EDT 上同步执行。invokeLater()方法应该在一个应用程序线程需要更新 GUI 时才被使用。它可以从任意线程中被调用。
- boolean isEventDispatchThread()：当调用线程是 EDT 时，返回 true；否则，返回 false。

invokeAndWait()、invokeLater()以及 isEventDispatchThread()方法就是调用类 java.awt.EventQueue 中同等方法的包装器。尽管你给这些方法加上 SwingUtilities 前缀，而我则使用 EventQueue 作为前缀（出于习惯）。

你通常会根据以下模式使用 invokeLater()方法构建一个 Swing 的 GUI：

```
Runnable r = ... // ... refers to the runnable's anonymous class or lambda
EventQueue.invokeLater(r);
```

清单 B-2 展示了第 2 版 ViewPage 的源代码，它使用了 invokeLater()方法在 EDT 之上构造了 Swing GUI。

清单 B-2　在 EDT 之上构造 HTML 查看器的 Swing 应用程序的 GUI

```
import java.awt.BorderLayout;
import java.awt.Dimension;
```

```
import java.awt.EventQueue;

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import java.io.InputStream;
import java.io.IOException;

import java.net.URL;

import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;
import javax.swing.JTextField;

public class ViewPage
{
   public static void main(String[] args)
   {
      Runnable r = () ->
      {
         final JFrame frame = new JFrame("View Page");
         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
         JPanel panel = new JPanel();
         panel.add(new JLabel("Enter URL"));
         final JTextField txtURL = new JTextField(40);
```

```
panel.add(txtURL);
frame.getContentPane().add(panel, BorderLayout.NORTH);
final JTextArea txtHTML = new JTextArea(10, 40);
frame.getContentPane().add(new JScrollPane (txtHTML),
                           BorderLayout.CENTER);
ActionListener al = (ae) ->
{
   InputStream is = null;
   try
   {
      URL url = new URL(txtURL.getText());
      is = url.openStream();
      StringBuilder sb = new StringBuilder();
      int b;
      while ((b = is.read()) != -1)
         sb.append((char) b);
      txtHTML.setText(sb.toString());
   }
   catch (IOException ioe)
   {
      txtHTML.setText(ioe.getMessage());
   }
   finally
   {
      txtHTML.setCaretPosition(0);
      if (is != null)
         try
         {
```

```
                    is.close();
                }
                catch (IOException ioe)
                {
                }
            }
        };
        txtURL.addActionListener(al);
        frame.pack();
        frame.setVisible(true);
    };
    EventQueue.invokeLater(r);
  }
}
```

清单 B-2 解决了一个问题，但是我们依旧得防止 EDT 被延迟。我们可以通过在 EDT 之上创建一条工作线程去读取页面，而后使用 `invokeAndWait()` 方法把页面内容更新到滚动的文本区域来解决这一问题。

清单 B-3 在非延迟的 EDT 之上构造 HTML 查看器的 Swing 应用程序的 GUI

```
import java.awt.BorderLayout;
import java.awt.Dimension;
import java.awt.EventQueue;

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import java.io.InputStream;
import java.io.IOException;
```

```
import java.lang.reflect.InvocationTargetException;

import java.net.URL;

import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;
import javax.swing.JTextField;

public class ViewPage
{
   public static void main(String[] args)
   {
      Runnable r = () ->
      {
         final JFrame frame = new JFrame("View Page");
         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
         JPanel panel = new JPanel();
         panel.add(new JLabel("Enter URL"));
         final JTextField txtURL = new JTextField(40);
         panel.add(txtURL);
         frame.getContentPane().add(panel, BorderLayout.NORTH);
         final JTextArea txtHTML = new JTextArea(10, 40);
         frame.getContentPane().add(new JScrollPane (txtHTML),
                                    BorderLayout.CENTER);
```

```
ActionListener al = (ae) ->
{
   txtURL.setEnabled(false);
   Runnable worker = () ->
   {
      InputStream is = null;
      try
      {
         URL url = new URL(txtURL.getText());
         is = url.openStream();
         final StringBuilder sb = new StringBuilder();
         int b;
         while ((b = is.read()) != -1)
            sb.append((char) b);
         Runnable r1 = () ->
         {
            txtHTML.setText(sb.toString());
            txtURL.setEnabled(true);
         };
         try
         {
            EventQueue.invokeAndWait(r1);
         }
         catch (InterruptedException ie)
         {
         }
         catch (InvocationTargetException ite)
         {
```

```
            }
        }
        catch (final IOException ioe)
        {
            Runnable r1 = () ->
            {
                txtHTML.setText(ioe.getMessage());
                txtURL.setEnabled(true);
            };
            try
            {
                EventQueue.invokeAndWait(r1);
            }
            catch (InterruptedException ie)
            {
            }
            catch (InvocationTargetException ite)
            {
            }
        }
        finally
        {
            Runnable r1 = () ->
            {
                txtHTML.setCaretPosition(0);
                txtURL.setEnabled(true);
            };
            try
```

```
                {
                    EventQueue.invokeAndWait(r1);
                }
                catch (InterruptedException ie)
                {
                }
                catch (InvocationTargetException ite)
                {
                }
                if (is != null)
                    try
                    {
                        is.close();
                    }
                    catch (IOException ioe)
                    {
                    }
            }
        };
        new Thread(worker).start();
    };
    txtURL.addActionListener(al);
    frame.pack();
    frame.setVisible(true);
};
EventQueue.invokeLater(r);
}
}
```

我在获取页面时禁用文本框而后再启用。而你仍然可以在任意时刻关闭 GUI。

尽管清单 B-3 解决了 GUI 无响应的问题，但是解决方案还是稍微冗余了点。好在，这儿有替代方案。

B.2.2 SwingWorker

Swing 提供了类 `javax.swing.SwingWorker` 以较少的冗余来容纳长时间运行的任务（如读取 URL 的内容）。你必须继承这个抽象类并重写一个或多个方法来有效地完成工作。

SwingWorker 的泛型类型是 `SwingWorker<T, V>`。参数 `T` 和 `V` 区分了最终以及中间任务的结果类型。

你可以重写 `protected abstract T doInBackground()`方法在一条工作线程中执行长时间的任务并且返回一个 `T` 类型的结果（当没有结果时，返回 `Void` 类型）。当该方法结束，`protected void done()`方法就会在 EDT 上被调用。默认情况下，该方法不做任何事。不过，你也可以重写 `done()`方法来安全地更新 GUI。

当任务在运行时，你可以通过调用 `protected void publish(V... chunks)`方法周期性地将结果发布到 EDT 中。这些结果会被在 EDT 中运行的重写过的 `protected void process(List<V> chunks)`方法获取到。如果没有中间结果需要处理，你可以为 `V` 指定 `Void` 类型（或者不使用 `publish()`及 `process()`方法）。

`SwingWorker` 有两个及以上的方法需要你了解。首先，`void execute()`方法在一条工作线程调度被调用的 `SwingWorker` 对象。其次，`T get()`方法在需要时会等待 `doInBackground()`方法完成，而后返回最终结果。

注意：尝试获取从 doInBackground()方法中获取对象过程中如果发生异常，那么 SwingWorker 的 get()方法就会抛出类 java.util.concurrent.ExecutionException 的实例。它也可以抛出 InterruptedException。

清单 B-4 展示了最终 ViewPage 应用程序的源码，它使用 SwingWorker 取代了 invokeAndWait()方法。

清单 B-4 重新在非延迟的 EDT 之上构造 HTML 查看器的 Swing 应用程序的 GUI

```
import java.awt.BorderLayout;
import java.awt.Dimension;
import java.awt.EventQueue;

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import java.io.InputStream;
import java.io.IOException;

import java.net.URL;

import java.util.concurrent.ExecutionException;

import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;
```

```
import javax.swing.JTextField;
import javax.swing.SwingWorker;

public class ViewPage
{
   public static void main(String[] args)
   {
      Runnable r = () ->
      {
         final JFrame frame = new JFrame("View Page");
         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
         JPanel panel = new JPanel();
         panel.add(new JLabel("Enter URL"));
         final JTextField txtURL = new JTextField(40);
         panel.add(txtURL);
         frame.getContentPane().add(panel, BorderLayout.NORTH);
         final JTextArea txtHTML = new JTextArea(10, 40);
         frame.getContentPane().add(new JScrollPane (txtHTML),
                                    BorderLayout.CENTER);
         ActionListener al = (ae) ->
         {
            txtURL.setEnabled(false);
            class GetHTML extends SwingWorker<StringBuilder, Void>
            {
               private final String url;

               GetHTML(String url)
               {
```

```
        this.url = url;
    }

    @Override
    public StringBuilder doInBackground()
    {
        StringBuilder sb = new StringBuilder();
        InputStream is = null;
        try
        {
            URL url = new URL(this.url);
            is = url.openStream();
            int b;
            while ((b = is.read()) != -1)
                sb.append((char) b);
            return sb;
        }
        catch (IOException ioe)
        {
            sb.setLength(0);
            sb.append(ioe.getMessage());
            return sb;
        }
        finally
        {
            if (is != null)
                try
                {
```

```
                    is.close();
                }
                catch (IOException ioe)
                {
                }
            }
        }

        @Override
        public void done()
        {
            try
            {
                StringBuilder sb = get();
                txtHTML.setText(sb.toString());
                txtHTML.setCaretPosition(0);
            }
            catch (ExecutionException ee)
            {
                txtHTML.setText(ee.getMessage());
            }
            catch (InterruptedException ie)
            {
                txtHTML.setText("Interrupted");
            }
            txtURL.setEnabled(true);
        }
    }
```

```
            new GetHTML(txtURL.getText()).execute();
        };
        txtURL.addActionListener(al);
        frame.pack();
        frame.setVisible(true);
    };
    EventQueue.invokeLater(r);
  }
}
```

最终版本的 ViewPage 依赖于 GetHTML，这是在动作监听器 lambda 表达式中声明的本地 SwingWorker 子类，来在工作线程中读取 Web 页面（保持用户持续响应），然后用 HTML 更新 EDT（Swing 的代码必须在这里执行）之上的用户界面。

当 lambda 表达式运行时(用户在文本框中输入一个 URL 之后按下回车键)，它用文本框中的文本初始化了 GetHTML 类（由于 Swing 是单线程的，无法从工作线程中访问文本框），并且调用了 SwingWorker 的 execute()方法。

execute()方法导致 GetHTML 中重写的 doInBackground()方法在工作线程中被调用，这会产生并返回一个包裹 HTML 或者错误文本的 java.lang.StringBuilder 对象。EDT 之后会调用重写的 done()方法，该方法会调用 SwingWorker 的 get()方法访问这个 StringBuilder 对象，并且把内容输出到文本区域。

B.2.3 定时器

Swing 提供了类 javax.swing.Timer（很像一个简化版的定时器框架，请见第 4 章）在 EDT 上周期性地执行 Swing 的代码。它会在一个初始的延时之

后触发一个动作事件到注册过的监听器当中，并且此后以固定的事件间隔重复触发。

调用构造函数 `Timer(int delay, ActionListener listener)`创建一个有初始延时以及事件间隔延时的定时器，并且每隔 `delay` 毫秒，事件就会被发送到初始的动作监听器（可能为空）中。

`delay` 这个参数值会用于初始延时以及固定的事件间隔。当然你也可以分别将这些值通过 `void setInitialDelay(int initialDelay)` 和 `void setDelay(int delay)`方法设置进去。

注意：用 `false` 作为参数调用 `Timer` 的 `void setRepeats(boolean flag)` 方法，会导致该定时器仅发送一条动作事件。

调用 `void addActionListener(ActionListener listener)`方法可以添加另外一个动作监听器，而调用 `void removeActionListener(ActionListener listener)`方法则移除了之前注册的动作监听器。调用 `ActionListener[] getActionListeners()`方法会获取所有注册的监听器。

新创建的定时器处于停止状态。为了启动这个定时器，需要调用其 `void start()`方法。相对地，你需要调用 `void stop()`方法来终止这个定时器。你或许也想要调用 `boolean isRunning()`方法来确定这个定时器是否正在运行。

清单 B-5 给出了一个计数器应用程序的源码，它会创建一个定时器来持续地在一个标签上显示跳动的计数。

清单 B-5　开始和停止计数

```
import java.awt.EventQueue;
import java.awt.FlowLayout;
```

```
import java.awt.event.ActionListener;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.Timer;

public class Counter extends JFrame
{
   int count;

   public Counter(String title)
   {
      super(title);
      setDefaultCloseOperation(EXIT_ON_CLOSE);

      JPanel pnl = new JPanel();
      ((FlowLayout) pnl.getLayout()).setHgap(20);
      final JLabel lblCount = new JLabel("");
      pnl.add(lblCount);
      final JButton btnStartStop = new JButton("Start");
      ActionListener al = (ae) ->
      {
         ++count;
         lblCount.setText(count + " ");
      };
```

```
        final Timer timer = new Timer(30, al);
        al = (ae) ->
        {
           if (btnStartStop.getText().equals("Start"))
           {
              btnStartStop.setText("Stop");
              timer.start();
           }
           else
           {
              btnStartStop.setText("Start");
              timer.stop();
           }
        };
        btnStartStop.addActionListener(al);
        pnl.add(btnStartStop);
        setContentPane(pnl);

        setSize(300, 80);
        setVisible(true);
     }

     public static void main(String[] args)
     {
        EventQueue.invokeLater(() -> new Counter("Counter"));
     }
  }
```

清单 B-5 中 main()方法创建一个由标签和开始、停止按钮构成的 GUI。这个标签展示 count 变量的当前值，而按钮上的文字则在开始和停止之间切换。当按钮显示开始时，点击会触发定时器启动；而按钮显示停止时，点击会导致定时器停止。定时器的动作监听器递增 count 变量并将值显示到标签上。添加在 count 变量之后的空格字符串，会把表达式转换成一个字符串，从而保证不会截断最右边的像素。

照下面编译清单 B-5：

```
javac Counter.java
```

运行程序：

```
java Counter
```

图 B-2 展示了结果 GUI。

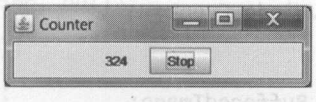


图 B-2　面板中的组件水平居中

B.3　基于定时器的幻灯片展示

幻灯片展示就是在投影仪屏幕之上演示一组静态图片，通常会以预先安排好的顺序播放。每张图片一般都会至少显示几秒而后被下一张图片替换。

幻灯片的展示会涉及一个投影仪、一个屏幕和一组幻灯片。投影仪含有将要投影的幻灯片，屏幕用于展示被投影的幻灯片，而幻灯片则包含一张图片和其他属性（例如一个文本的标题）。

我创建了一个名为 SlideShow 的 Java 应用程序，它可以让你投影任意的幻灯片。清单 B-6 是它的源码。

清单 B-6 描述一个基于定时器的幻灯片展示

```
import java.awt.AlphaComposite;
import java.awt.Color;
import java.awt.Dimension;
import java.awt.EventQueue;
import java.awt.Font;
import java.awt.FontMetrics;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.RenderingHints;
import java.awt.event.ActionListener;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;

import java.awt.image.BufferedImage;

import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;
import java.io.IOException;

import java.util.ArrayList;
import java.util.List;

import javax.imageio.ImageIO;

import javax.swing.JComponent;
```

```
import javax.swing.JFrame;
import javax.swing.Timer;

class Projector
{
   private volatile List<Slide> slides;
   private Screen s;
   private Timer t;
   private volatile int slideIndexC, slideIndexN;
   private volatile float weight;

   Projector(List<Slide> slides, Screen s)
   {
      this.slides = slides;
      this.s = s;
      t = new Timer(1500, null);
      t.setDelay(3000);
      slideIndexC = 0;
      slideIndexN = 1;
   }

   void start()
   {
      s.drawImage(Slide.blend(slides.get(0), null, 1.0f));
      ActionListener al = (ae) ->
      {
         weight = 1.0f;
         Timer t2 = new Timer(0, null);
         t2.setDelay(10);
         ActionListener al2 = (ae2) ->
```

```
            {
               Slide slideC = slides.get(slideIndexC);
               Slide slideN = slides.get(slideIndexN);
               BufferedImage bi = Slide.blend(slideC, slideN, weight);
               s.drawImage(bi);
               weight -= 0.01f;
               if (weight <= 0.0f)
               {
                  t2.stop();
                  slideIndexC = slideIndexN;
                  slideIndexN = (slideIndexN + 1) % slides.size();
               }
            };
            t2.addActionListener(al2);
            t2.start();
         };
         t.addActionListener(al);
         t.start();
      }

      void stop()
      {
         t.stop();
      }
   }

   class Screen extends JComponent
   {
      private Dimension d;
      private BufferedImage bi;
```

```
    private String text;

    Screen(int width, int height)
    {
       d = new Dimension(width, height);
    }

    void drawImage(BufferedImage bi)
    {
       this.bi = bi;
       repaint();
    }

    @Override
    public Dimension getPreferredSize()
    {
       return d;
    }

    @Override
    public void paint(Graphics g)
    {
       int w = getWidth();
       int h = getHeight();
       g.drawImage(bi, Slide.WIDTH <= w ? (w - Slide.WIDTH) / 2 : 0,
                   Slide.HEIGHT <= h ? (h - Slide.HEIGHT) / 2 : 0, null);
    }
 }

 class Slide
```

```
{
    static int WIDTH, HEIGHT;

    private static int TEXTBOX_WIDTH, TEXTBOX_HEIGHT, TEXTBOX_X, TEXTBOX_Y;

    private BufferedImage bi;
    private String text;
    private static Font font;

    private Slide(BufferedImage bi, String text)
    {
        this.bi = bi;
        this.text = text;
        font = new Font("Arial", Font.BOLD, 20);
    }

    static BufferedImage blend(Slide slide1, Slide slide2, float weight)
    {
        BufferedImage bi1 = slide1.getBufferedImage();
        BufferedImage bi2 = (slide2 != null)
                            ? slide2.getBufferedImage()
                            : new BufferedImage(Slide.WIDTH, Slide.HEIGHT,
                                                BufferedImage.TYPE_INT_RGB);
        BufferedImage bi3 = new BufferedImage(Slide.WIDTH, Slide.HEIGHT,
                                              BufferedImage.TYPE_INT_RGB);
        Graphics2D g2d = bi3.createGraphics();
        g2d.setComposite(AlphaComposite.getInstance(AlphaComposite.SRC_OVER,
                                                    weight));
        g2d.drawImage(bi1, 0, 0, null);
        g2d.setComposite(AlphaComposite.getInstance(AlphaComposite.SRC_OVER,
```

```
                                                    1.0f - weight));
    g2d.drawImage(bi2, 0, 0, null);
    g2d.setColor(Color.BLACK);
    g2d.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
                         RenderingHints.VALUE_ANTIALIAS_ON);
    g2d.setComposite(AlphaComposite.getInstance(AlphaComposite.SRC_OVER,
                                                0.5f));
    g2d.fillRect(TEXTBOX_X, TEXTBOX_Y, TEXTBOX_WIDTH, TEXTBOX_HEIGHT);
    g2d.setComposite(AlphaComposite.getInstance(AlphaComposite.SRC_OVER,
                                                weight));
    g2d.setColor(Color.WHITE);
    g2d.setFont(font);
    FontMetrics fm = g2d.getFontMetrics();
    g2d.drawString(slide1.getText(), TEXTBOX_X + (TEXTBOX_WIDTH -
                   fm.stringWidth(slide1.getText())) / 2,
                   TEXTBOX_Y + TEXTBOX_HEIGHT / 2 + fm.getHeight() / 4);
    g2d.setComposite(AlphaComposite.getInstance(AlphaComposite.SRC_OVER,
                                                1.0f - weight));
    if (slide2 != null)
      g2d.drawString(slide2.getText(), TEXTBOX_X + (TEXTBOX_WIDTH -
                     fm.stringWidth(slide2.getText())) / 2, TEXTBOX_Y +
                     TEXTBOX_HEIGHT / 2 + fm.getHeight() / 4);
    g2d.dispose();
    return bi3;
  }

  BufferedImage getBufferedImage()
  {
    return bi;
```

```
}

String getText()
{
   return text;
}

static List<Slide> loadSlides(String imagesPath) throws IOException
{
   File imageFilesPath = new File(imagesPath);
   if (!imageFilesPath.isDirectory())
      throw new IOException(imagesPath + " identifies a file");
   List<Slide> slides = new ArrayList<>();
   try (FileReader fr = new FileReader(imagesPath + "/index");
      BufferedReader br = new BufferedReader(fr))
   {
      String line;
      while ((line = br.readLine()) != null)
      {
         String[] parts = line.split(",");
         File file = new File(imageFilesPath + "/" + parts[0] + ".jpg");
         System.out.println(file);
         BufferedImage bi = ImageIO.read(file);
         if (WIDTH == 0)
         {
            WIDTH = bi.getWidth();
            HEIGHT = bi.getHeight();
            TEXTBOX_WIDTH = WIDTH / 2 + 10;
            TEXTBOX_HEIGHT = HEIGHT / 10;
```

```
                TEXTBOX_Y = HEIGHT - TEXTBOX_HEIGHT - 5;
                TEXTBOX_X = (WIDTH - TEXTBOX_WIDTH) / 2;
            }
            slides.add(new Slide(bi, parts[1]));
         }
      }
      if (slides.size() < 2)
         throw new IOException("at least one image must be loaded");
      return slides;
   }
}

public class SlideShow
{
   public static void main(String[] args) throws IOException
   {
      if (args.length != 1)
      {
         System.err.println("usage: java SlideShow ssdir");
         return;
      }
      List<Slide> slides = Slide.loadSlides(args[0]);
      final Screen screen = new Screen(Slide.WIDTH, Slide.HEIGHT);
      final Projector p = new Projector(slides, screen);
      Runnable r = () ->
      {
         final JFrame f = new JFrame("Slide Show");
         WindowAdapter wa = new WindowAdapter()
               {
                  @Override
```

```
                public void windowClosing(WindowEvent we)
                {
                    p.stop();
                    f.dispose();
                }
            };
            f.addWindowListener(wa);
            f.setContentPane(screen);
            f.pack();
            f.setVisible(true);
            p.start();
        };
        EventQueue.invokeLater(r);
    }
}
```

清单 B-6 基于类 Projector、Screen、Slide 以及 SlideShow 构建了一个幻灯片展示。Projector 声明了几个 private 属性、一个用一组(java.util.List) Slide 对象和单个 Screen 对象初始化投影仪对象的 Projector(List slides, Screen s)构造函数、一个用于启动投影仪的 void start()方法和一个停止投影仪的 stop()方法。

Screen 继承自 javax.swing.JComponent，这使得 Screen 变成了一个特殊类型的 Swing 组件。它声明了几个 private 属性、一个接收 width 和 height 参数来初始化该组件的 Screen(int width, int height)构造函数，以及一个 drawImage(BufferedImage bi)方法，用于在屏幕的表面绘制传递进来的缓冲图片。这个类也重写了 Dimension getPreferredSize()和 void paint (Graphics g)方法，它们分别用于返回该组件的偏好大小和绘制图片。

Slide 声明了多个常量、几个私有属性，一个用于初始化 Slide 对象的 Slide(BufferedImage bi, String text)的私有构造函数、两个用于返回幻灯片的缓冲图片和文字的 getter 方法 BufferedImagegetBuffered Image()和 String getText()、一个用于混合一对缓冲图片来实现幻灯片之间的过渡效果的类方法 BufferedImage blend(Slide slide1, Slide slide2, float weight)，以及用于加载所有幻灯片图片的方法 List loadSlides(String imagesPath)。

blend()方法抽取了与其 slide 参数相关联的缓冲图片，并且用 weight 值（必须落在 0.0 到 1.0 这个区间内）决定的混合度把这些图片混合到一起。传递给 weight 的值越高，slide1 的图片就会更多地融入返回的缓冲图片中。混合图片之后，blend()方法在已混合的图片之上再混入一对文字字符串。类 java.awt.AlphaComposite 参与到每次混合操作中。

我已经设计 blend()方法处理 slide2 是 null 的特殊场景。这会发生在 Projector 的 start()方法开始的时候，这时它执行了 s.drawImage (Slide.blend(slides.get(0), null, 1.0f));方法来显示第一张幻灯片，并且不会有过渡。

loadSlides()方法查找字符串参数所指定目录底下名为 index 的文本文件，然后创建一组被文本文件排序的 Slides——你可以选择一种不同于该目录下文件所确定的图片顺序来展示幻灯片。文本文件中的每一行都被组织成一个文件名后跟着一个分号，再跟着一个文字的描述（例如，**earth、Terran System**）。在指定文件名字的时候，你不必指定文件的扩展名，loadSlides()方法只能识别 JPEG 文件。

SlideShow 声明了一个 main()方法用于驱动这个应用程序。这个方法首先验证一个单一的命令行参数以确定幻灯片的目录（目录中包含了 index 和

JPEG 文件）已经就绪。之后它调用了 loadSlides()方法从这个目录中加载 index 和所有幻灯片图片。当 loadSlides()方法无法加载图片或者图片的数量小于 2 时，它抛出 java.io.IOException。毕竟，你如何用少于 2 张的图片来做幻灯片演示呢？

main()方法接下来创建一个 Screen 组件对象用于展示幻灯片图片。它把每个 slide（事实上，是每个幻灯片图片的宽度和高度）的宽度和高度传递给 Screen 的构造函数，保证了屏幕能够充分展示这些幻灯片（尽管我在 loadSlides()方法中没做强制，但是所有的幻灯片图片必须用于同样的宽度和高度）。

剩下唯一需要创建的主要模型对象就是这个投影仪，main()方法把一组从 loadSlides()方法中返回的 Slide 对象和前面创建的 Screen 对象传递到 Projector 的构造函数中，从而完成了这一任务。

main()方法最后的任务就是触发 GUI 能在 EDT 上构造出来。这个线程把内容面板设置到 Screen 对象中，并且调用了 Projector 的 void start()方法开始播放幻灯片。同时，它也创建了一个窗口监听器，当用户试图关闭窗口时，用于调用 Projector 的 void stop()方法。之后窗口就会被销毁掉。

Projector 使用了一对 Timer 对象来管理幻灯片演示。主定时器负责推进投影仪进入下一张幻灯片，而次定时器对象（在主定时器每次触发一个动作事件时被创建出来）则负责过渡当前播放的幻灯片到下一张（借助 blend()方法）。

每个定时器实例都运行在 EDT 之上。次定时器在运行时，主定时器不可以执行定时器任务。如果打破这个规则，这个幻灯片演示就会发生故障。我在主定时器任务相继执行之间选取了 3000 毫秒，而次定时器任务相继执行之间选取了 10 毫秒，这样运行 100 次，大约需要 1000 毫秒。当次定时器结束了，它会自动停止。

照下面编译清单 B-6：

```
javac SlideShow.java
```

假如是 Windows 操作系统，运行程序如下：

```
java SlideShow ..\ss
```

ss 标志这是一个关于太阳系的幻灯片演示（包含在本书的代码当中），放置于当前目录的父级目录下。

图 B-3 展示结果 GUI。



图 B-3 SlideShow 靠近幻灯片底部的文字水平居中

Apress®

Java线程API和并发工具的实用指南

Java线程与并发编程实践

Java的线程API和并发工具类，是Java自身强大而富有挑战性的API及语言特性的一部分。Java新手往往很难用这些特性编写出正确的多线程应用程序。本书旨在帮助Java开发者掌握并高效地使用这些功能。

本书分为两个部分，每部分包含4个章节。 第1部分关注于底层的线程API，而第2部分则关注于高级的并发工具类。第1部分介绍线程API的基础知识、同步、等待/通知以及线程组的额外能力，还有线程局部变量和定时器框架。第2部分介绍并发工具类的基础知识、Executor、同步器、锁框架，以及并发集合的额外能力、原子变量、Fork/Join框架和Completion Service。

每一章最后都给出了一些练习题。这些练习题旨在测试你对章节内容的理解程度。附录A提供这些习题的答案。附录B介绍了如何在Swing图形用户接口工具集(Swing Graphical User Interface Tookit)中使用线程。

通过本书，你将学习：

- 创建、配置并启动运行runnable的线程；
- 同步共享代码以避免竞态条件、数据竞争以及其他并发问题；
- 避免缓存变量引发的问题；
- 使用等待/通知机制协调多条线程；
- 探索线程组并学习避免使用它们的原因；
- 学习线程局部变量；
- 探索定时器框架；
- 学习引入并发工具类的原因；
- 探索Executor、同步器以及锁框架；
- 探索并发集合、原子变量、Fork/Join框架以及Completion Service；
- 实验Swing中的线程使用方式及其线程相关的API。

异步社区
人民邮电出版社
www.epubit.com.cn

异步社区 www.epubit.com.cn
新浪微博 @人邮异步社区
投稿/反馈邮箱 contact@epubit.com.cn

美术编辑：董志桢

分类建议：计算机／程序设计／Java
人民邮电出版社网址：www.ptpress.com.cn

ISBN 978-7-115-44036-5

ISBN 978-7-115-44036-5
定价：55.00 元